MASTER EN SCIENCES INFORMATIQUES

Conception et réalisation d'un noyau multitâche pour le processeur 8086

Henneau, Philippe

Award date:
1991

Awarding institution:
Universite de Namur

Facultés Universitaires
Notre Dame de la Paix.

5000 NAMUR

Institut d'informatique.

Conception et réalisation d'un noyau multitâche pour le processeur 8086.

par Philippe HENNEAU

Promoteur:

Professeur J. Ramaekers

Mémoire présenté en vue de
l'obtention du titre de licencié
et maître en informatique.

Année académique 1990-1991

La première partie de ce mémoire rappellera les notions de bases inhérentes aux systèmes multitâche monoprocesseur. Le concept de tâche en sera dégagé afin d'introduire l'ordonnancement et le module du noyau qui en est responsable. Ensuite, la pluralité des tâches nous permettra l'étude des notions spécifiques à la programmation concurrente: la compétition et la coopération entre tâches.

Au terme de ce rappel théorique, le lecteur aura acquis le bagage nécessaire à la compréhension de la deuxième partie: la conception et la réalisation en langage C de MYOS, mon exécutif multitâche. Celui-ci sera composé d'une part, d'un noyau minimal relatif à l'ordonnancement des tâches et , d'autre part, de primitives permettant de gérer la coopération et la compétition entre les tâches.

Enfin, en guise d'illustration, nous utiliserons MYOS pour développer dans la troisième partie quelques applications multitâche.

The first part of this thesis consists of a recall of the basic notions inherent to a monoprocessor multitasking system. Task concept will emerge from that analysis in order to explain the task switching. Next, we will study specific notions related to a multitasking system: the competition and the cooperation between tasks.

After that recall, the reader owns all the elements to understand the second part: the conception and the realization of MYOS, my multitasking executive. That one consists of two parts: firstly the minimal kernel related to task scheduling and, secondly all the primitives concerning competition and cooperation.

To end, as illustration, we will use MYOS to write some concurrent applications in the third part.

Remerciements.

Je tiens à remercier les membres du personnel de la firme SCHINDLER pour l'entourage et l'aide qu'ils m'ont apporté durant mon stage à Bruxelles:

Direction:

Ernst G.
Vancutsem J.

Laboratoire:

Aslanidis I.
Braye F.
Dams L.
Dehon G.
Delhauwe S. et son équipe
Eeckhout A.
Goffaux J-M.
Hermans C.
Heymans W.
Jousse B.
Leone S.
Lunardi M.
Matton K.
Moreau L.
Nottebart D.
Plees R.

Je ne voudrais pas oublier mes amis Christian Laloux et Nadine Dierick pour le dévouement qu'ils ont toujours su me témoigner.

Merci à tous.

Table des matières.

Introduction

Je me suis fixé comme objectif de concevoir et de réaliser un noyau multitâche pour le processeur 8086 équipant les pc-compatibles. Pour atteindre cet objectif, je décomposerai ce mémoire en trois parties:

dans la première je rappellerai les fondements théoriques relatifs aux systèmes multitâche monoprocesseur. C'est la concurrence qui retiendra d'abord mon attention. Elle me permettra d'introduire la notion de tâche et la commutation de tâches. Ensuite j'étudierai en détail la solution au problème de la compétition. J'aborderai à cet égard les sections critiques et l'exclusion mutuelle. J'expliquerai également les deux mécanismes de coopération: la communication et la synchronisation entre tâches. Pour terminer cette partie, je sensibiliserai le lecteur aux systèmes temps réel en analysant les contraintes de temps imposées par certains processus.

J'ai ainsi défini ce que je vais réaliser dans la deuxième partie: MYOS, mon noyau multitâche. Après avoir justifié la création de celui-ci, je réaliserai ses primitives. Celles-ci formeront les cinq parties du noyau relatives à la gestion des tâches, du clavier, des délais, de la communication et de la synchronisation.

Au terme de cette deuxième partie, je disposerai d'un support pour développer des applications concurrentes. J' écrirai à cet effet cinq programmes en langage C . Le but sera d'une part, d'initier le lecteur à l'utilisation de MYOS et d'autre part, de le sensibiliser à la programmation concurrente et à ses pièges.

Les annexes concernent l'utilisation de MYOS.Elles décrivent chacun de ses services. Cette partie sera très utile pour l'utilisateur désireux de créer des applications concurrentes.

1ère Partie.

I. Introduction à la concurrence

I.1 La notion de concurrence.

I.1.A Première approche.

La plupart des programmes traditionnels (calcul de paye par exemple) s'exécutent en séquence et leurs résultats sont indépendants de la vitesse à laquelle ils s'exécutent. Nous les qualifierons de **séquentiels**. Pour introduire la notion de concurrence nous évoquerons la définition d' ANDRE SCHIPER[1]:

" *Nous appellerons un programme concurrent un programme non séquentiel. Cette brève définition n'explique toutefois pas grand-chose. Pour apporter des éléments de définition, considérons l'expression arithmétique (A+B) * (C+D). Une évaluation séquentielle de cette expression consiste à calculer d'abord (A+B), puis (C+D), et finalement le produit (A+B) * (C+D). Une évaluation concurrente de cette même expression consiste à calculer* ***simultanément*** *(A+B) et (C+D) (avant d'en calculer le produit).*

*L'avantage de l'évaluation concurrente est évident si nous disposons de deux processeurs. Un premier processeur évaluera (A+B) pendant que l'autre évalue (C+D); le temps total nécessaire à l'évaluation de l'expression (A+B) * (C+D) se trouve ainsi diminué. Que se passe-t-il si nous ne disposons que d'un processeur? Précisons d'abord qu'évaluer de façon concurrente (A+B) * (C+D) doit être interprété dans ce cas de la manière suivante: le processeur est utilisé une* ***fraction*** *de seconde pour évaluer (A+B), puis une fraction de seconde pour évaluer (C+D), puis une fraction de seconde pour continuer d'évaluer (A+B), et ainsi de suite. Y a-t-il un avantage à procéder de cette manière? La réponse est NON. La programmation concurrente est pourtant une technique importante, même dans le cas d'un seul processeur. Pour en saisir l'intérêt, il est nécessaire d'aborder le problème de la gestion des entrées-sorties (figure 1). La programmation concurrente permet en effet d'exploiter pleinement la possibilité qu'ont le processeur et les périphériques de travailler en parallèle."*

---

[1] A. SCHIPER, Programmation concurrente, presses polytechniques romandes, Lausanne, 1986.

Supposons que nous ayons à réaliser un programme de communication (figure 1) dont le but est de recopier toutes les données en entrée (A et B) respectivement sur les ports de sortie A' et B'. Une des solutions est la **pseudo-concurrence** simple utilisant un **rescheduling implicite** (figure 2.)

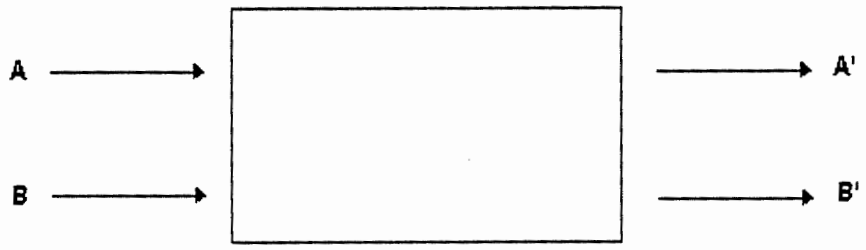


Figure 1: exemple simple de communication.

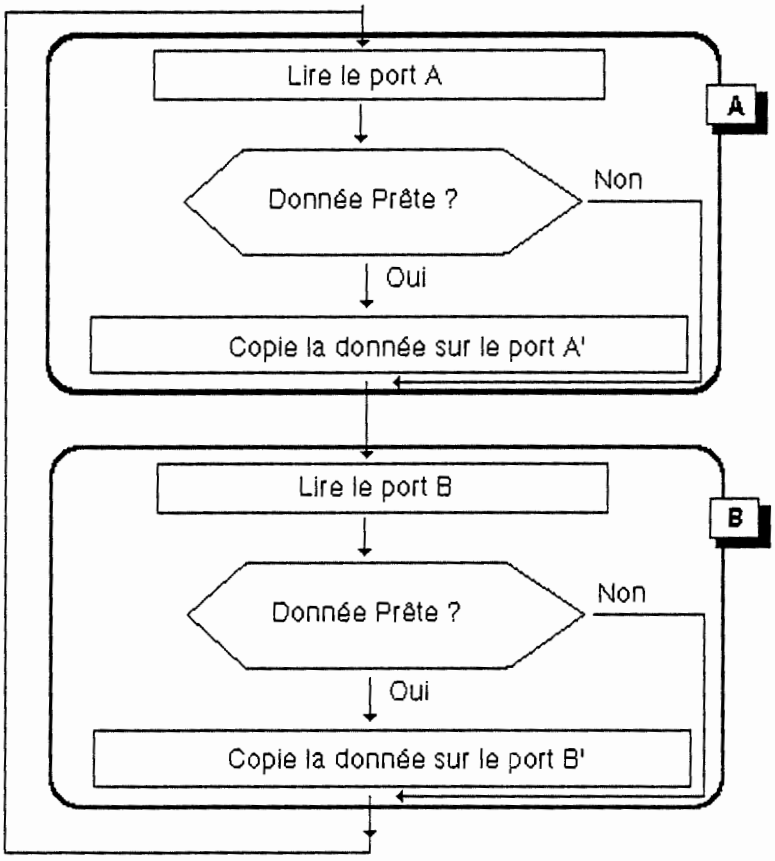


Figure 2: rescheduling implicite.

Supposons que l'on ajoute des fonctions non prévues à l'origine: une fonction mathématique à évaluer par exemple. Seulement, l'évaluation de cette fonction consomme trop de temps CPU ce qui ne permet plus une scrutation rapide des ports A et B. Une solution consiste à fractionner la fonction mathématique en plusieurs parties. A chaque cycle on évaluera une partie de la fonction comme suit:[2]

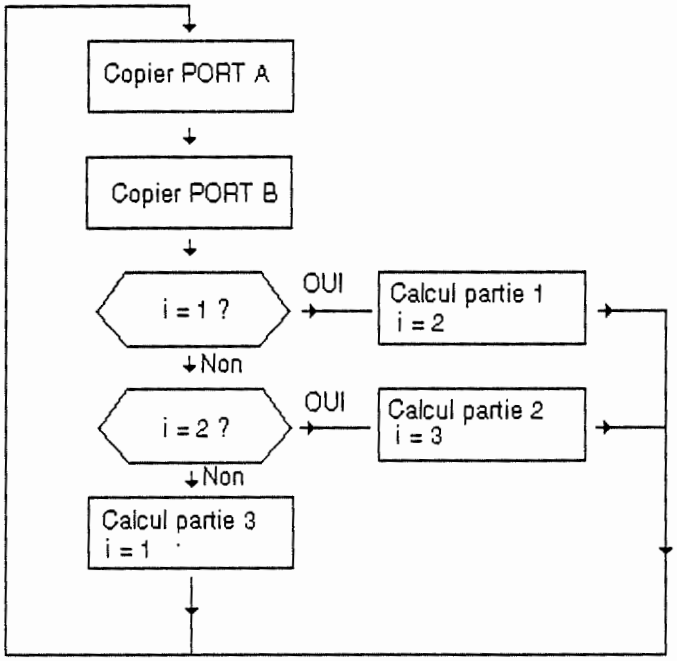


Figure 3: fractionnement de la fonction mathématique.

L'inconvénient de cette solution est que le programme doit être ajusté à chaque modification de la fonction mathématique.

[2]La fonction est ici décomposée en 3 parties

La **vraie concurrence** consisterait à disposer de plusieurs processeurs: la solution à notre problème deviendrait:

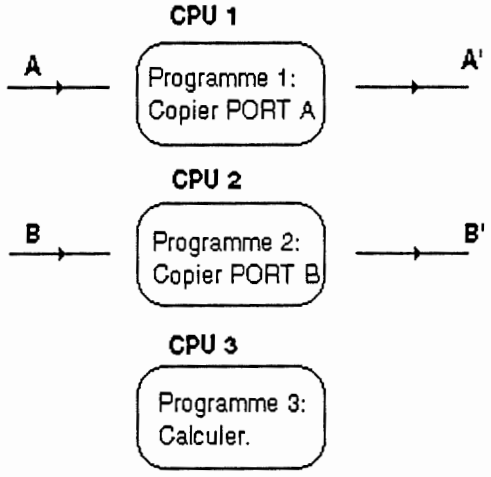


Figure 4: la vraie concurrence.

La présence de hardware supplémentaire élève considérablement le coût. Si nous restons équipés d'un seul processeur on **partagera** le **temps** entre différentes **tâches**, à chaque tâche correspondra un programme. La pseudo-concurrence permet ainsi la **simultanéité apparente** :

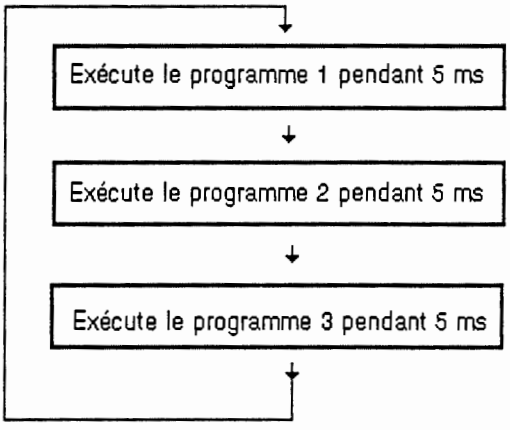


Figure 5: la pseudo-concurrence.

La solution ainsi obtenue est plus souple mais très inefficace: le programme 1 et le programme 2 utilisent les deux tiers du temps CPU même si aucune donnée n'arrive. Pour remédier à ce gaspillage de temps calcul, nous utiliserons les **interruptions** qui permettent d'attirer l'attention sur un événement extérieur évitant ainsi de scruter sans cesse les entrées. La solution à notre problème devient donc :

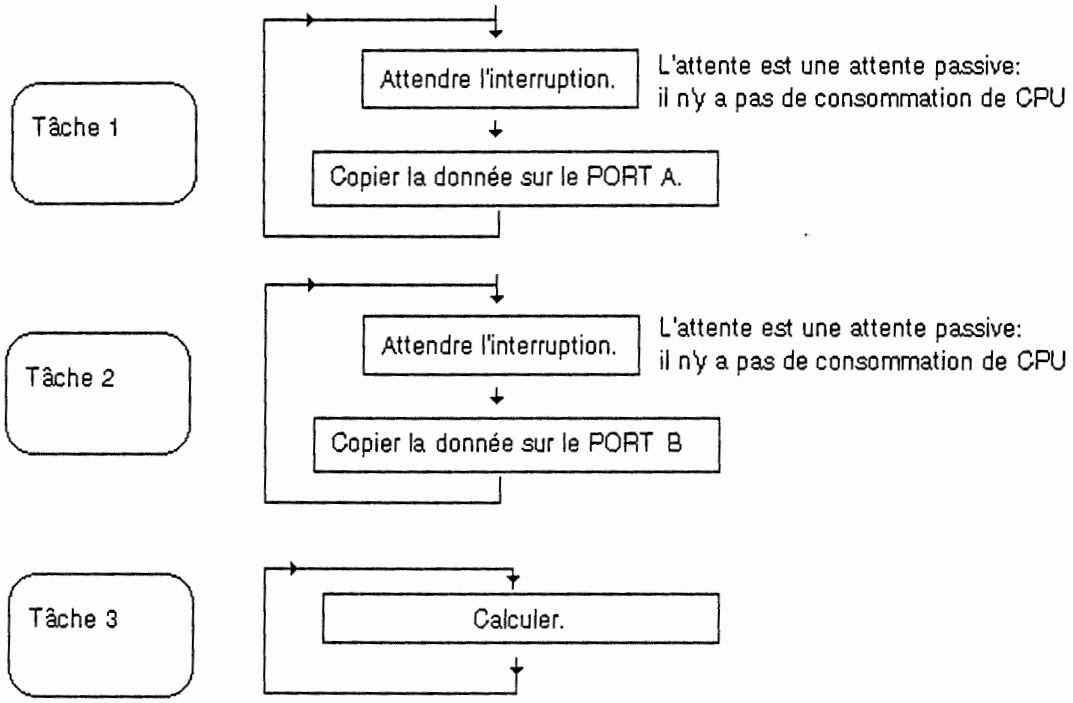


Figure 6: La notion d'événement et d'interruption.

Cette dernière solution est très performante mais génère un autre problème: la nécessité d'un **scheduler** ,programme allouant le processeur aux **tâches.**

I.1.B Définitions.

Application multitâche:

Capacité d'une application à gérer **simultanément**, de façon concurrente plusieurs tâches dont l'ensemble traduit la globalité du processus modélisé.

Processus:

Globalité des fonctionnalités du système physique modélisé ou sous contrôle.

Système préemptif:

Système capable de commuter les tâches sans que l'utilisateur l'ait prévu ou demandé. Entre deux instructions d'une tâche, d'autres instructions appartenant à une autre tâche peuvent être exécutées.

I.2 Modèle des tâches dans un système multitâche monoprocesseur.

I.2.A Le concept de tâche.

Une **tâche** est un programme ou une partie de programme en exécution. Théoriquement chaque tâche possède un **processeur virtuel** comprenant son pointeur d'instructions (IP), sa zone de donnée et son pointeur de pile SP (figure 7). En réalité, le processeur physique **commute** de tâches en tâches (figures 8 et 9) sous le contrôle d'un module particulier du noyau du système d'exploitation multitâche appelé **SCHEDULER.**[3]

Dans ce modèle les tâches ne doivent pas être programmées en faisant des hypothèses quant à leur durée d'exécution. Le programmeur n'a pas la maîtrise de l'attribution du processeur à une tâche, c'est le scheduler qui s'en acquitte.Comme pour la programmation monotâche, toute temporisation basée sur la notion de boucle doit être exclue, des instructions d'une autre tâche pouvant être exécutées entre celles de la boucle.

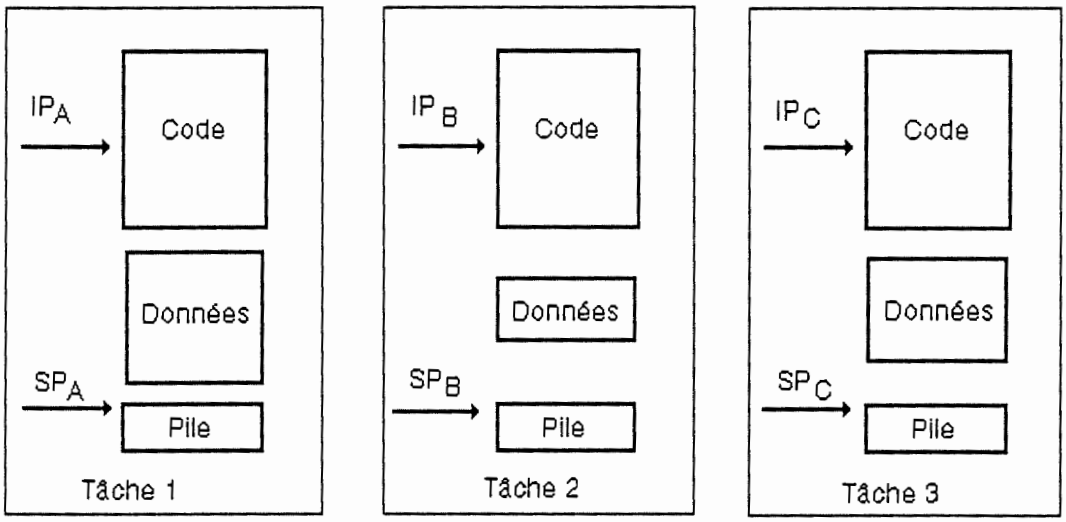


Figure 7: Modèle conceptuel d'un système comportant trois tâches.

[3] Celui-ci sera développé en I.3

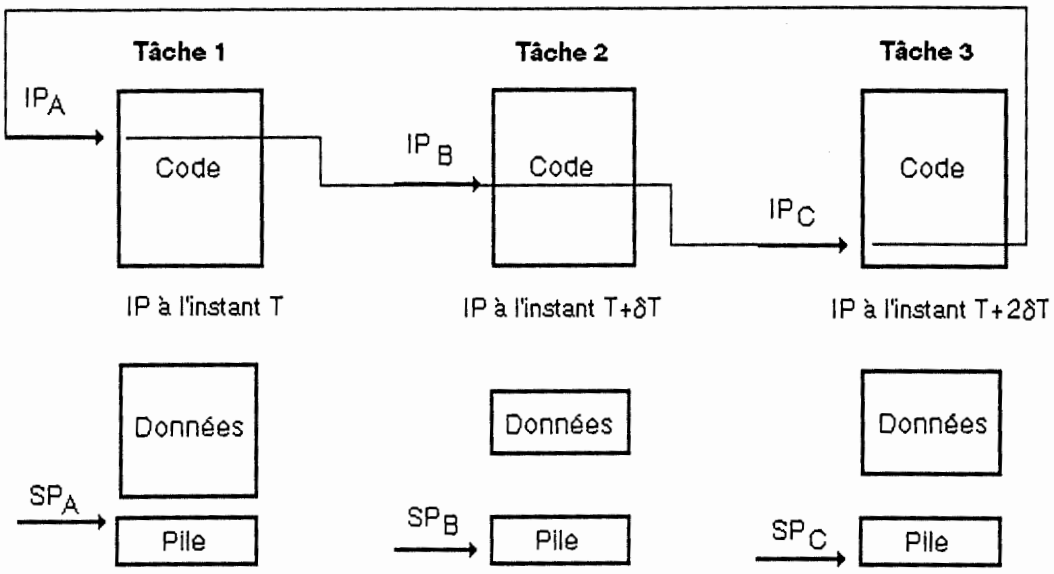


Figure 8: Modèle réel d'un système comportant trois tâches.

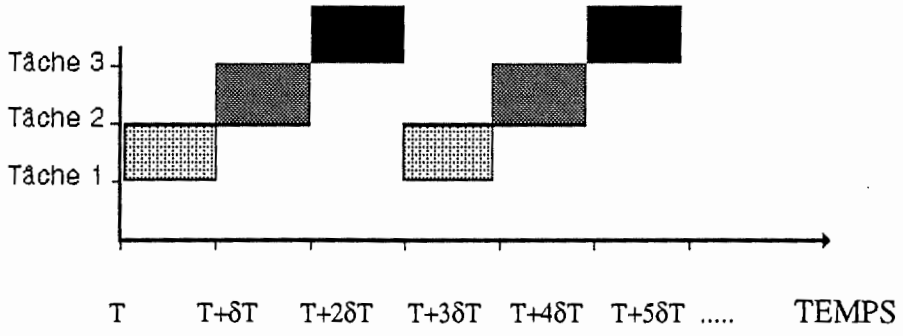


Figure 9: Occupation du processeur dans le temps pour trois tâches de même priorité

I.2.B Etats d'une tâche.[4]

Analysons les **états** d'une tâche et puis les **transitions** entre ces états.

Etat non créé

Une tâche dans l'état non créé est une tâche inconnue de l'exécutif. Dans un système comportant des périphériques de stockage ces tâches ne sont pas chargées en mémoire et ne possèdent pas de zone de code, de données ou de pile. Dans un système ne comportant pas ces dispositifs seul le code de la tâche est en mémoire.

Etat dormant ou état créé.

Lorsqu'une tâche est **connue** de l'exécutif, elle est dans l'état dormant: elle possède un identifiant,une pile et une adresse. Elle restera dans cet état tant qu'une requête du noyau ne la fait pas évoluer dans l'état prêt (création de tâche) ou dans l'état non créé (suppression de tâche.)

Etat prêt.

Une tâche est dans l'état prêt lorsqu'elle est **candidate** au processeur. Son exécution ne dépend que de sa **priorité** par rapport aux autres tâches prêtes ou en exécution[5]. Lorsqu'elle devient prioritaire, le scheduler lui octroie le processeur.

Etat en exécution.

La tâche dont le code est **exécuté** par le processeur est en exécution. Dans un système monoprocesseur il n'y a jamais qu'une seule tâche à la fois à être dans cet état.

---

[4] D. TSCHIRHART, commande en temps réel.

[5] Dans un système monoprocesseur il n'y a qu'une tâche à la fois en exécution (Cfr. état en exécution).

Etat suspendu.

Une condition de **blocage** a été rencontrée et la tâche s'est interrompue avant sa fin. Dans cet état elle est **ignorée** du scheduler, elle n'aura pas le processeur. Essentiellement, les conditions de blocage d'une tâche sont de deux types: suspension **explicite** (appel de la requête suspension de tâche du noyau) ou une suspension **implicite** (appel d'une fonction du noyau dont l'effet, dans certains cas, sera de suspendre la tâche: attente d'une sémaphore de libre, attente d'un message, attente d'événement ...)

I.2.C Transitions entre les états.

La plupart des transitions entre les états sont la conséquence d'un appel d'une fonction du noyau. Suivons la vie d'une tâche et examinons tous les états par lesquels elle peut transiter:

Création de la tâche.

La création d'une tâche consiste à la faire connaître du noyau. Durant cette opération, celui-ci reçoit l'adresse et la priorité de la tâche créée. En retour il réserve une pile et fournit un identifiant qui par la suite est le seul moyen d'identifier la tâche.

Activation de la tâche.

L'activation est une demande d'exécution. Après cette transition la tâche est dans l'état prêt.

Suspension d'une tâche.

La tâche est mise dans l'état suspendu et elle abandonne le processeur. Rappelons que cette transition est le résultat d'une suspension **explicite** ou **implicite**

Relance d'une tâche suspendue.

La condition de blocage est levée, la tâche est donc remise dans l'état prêt. Ici aussi la relance peut être **explicite** (appel d'une fonction de relance du noyau) ou

implicite (un message est arrivé à destination d'une tâche en attente d'un message, une sémaphore est libérée ...) La tâche est donc insérée dans une file d'attente pour l'obtention du processeur qu'elle obtiendra en fonction de sa priorité.

Destruction d'une tâche.

La tâche concernée est remise dans l'état dormant. Une tâche est détruite par un appel **explicite** d'une fonction du noyau ou lorsqu'elle est **terminée**[6].

Suppression d'une tâche

Pour optimiser l'usage de la mémoire une requête de suppression de tâche la remet dans l'état non-créé.

Le scheduling

La gestion de l'exécution des tâches candidates au processeur (donc dans l'état prêt) est assurée par le scheduler qui mérite une nouvelle section à lui seul!

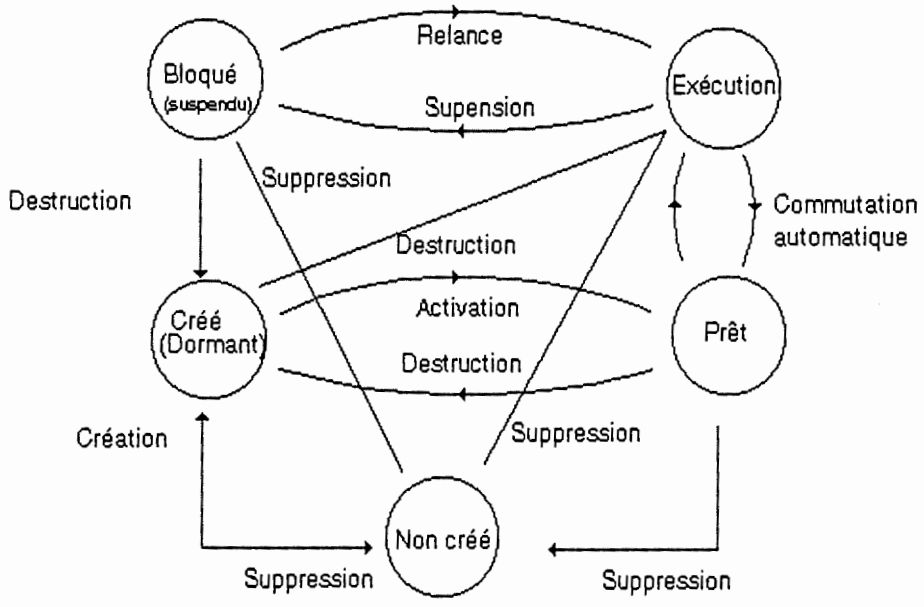


Figure 10: Etats et transitions.

[6] Lorsque sa dernière instruction a été exécutée.

I.3 Le scheduling (l'ordonnancement).

I.3.A Systèmes avec ou sans réquisition du processeur.

Les noyaux se divisent en deux catégories: les noyaux **sans réquisition** du processeur (no preemptive scheduling) et noyaux **avec réquisition** du processeur (préemptive scheduling).

Noyaux sans réquisition du processeur.

Dans ce genre de noyau, la tâche est exécutée jusqu'à ce qu'elle fasse **appel** à un service du noyau. Celui-ci examine alors les priorités des tâches dans l'état prêt. Si il existe au moins une tâche plus prioritaire, la tâche qui était en exécution perd le processeur au profit d'une plus prioritaire. Ce noyau présente un inconvénient non négligeable: si pour une raison quelconque une tâche boucle, tout le système est bloqué.

L'exécution dans ce genre de système est du **quasi-parallélisme**. Il est souvent utilisé pour des simulations. Les tâches dans un tel système portent le nom de coroutines.

Noyaux avec réquisition du processeur.

Ici , une tâche peut à tout instant **perdre** le processeur au profit d'une autre. La tâche qui perd le processeur n'a aucun moyen de le savoir. Dans ce genre de système, c'est une interruption de l'horloge qui active le scheduler. Celui-ci met la tâche en exécution dans l'état prêt, la place à la fin dans la file d'attente et exécute la première tâche de cette file (FIFO). L'exécution ici est du **pseudo-parallélisme**

I.3.B Le scheduler.

Le rôle du scheduler est de **commuter** les tâches. La commutation est le passage de l'exécution d'une tâche T1 à l'exécution d'une autre tâche T2. Chaque fois qu'une tâche est remise dans l'état prêt après avoir eu le processeur pendant un cycle, tout son **contexte** doit être sauvegardé: le stack, tous les registres et le pointeur d'instructions[7] pour qu'elle puisse continuer son exécution dans le même environnement .

Les interruptions 'hardware' permettent d'activer le scheduler au rythme de l'horloge du système[8]. Chaque tâche possède donc une quantité de temps ou **quantum** durant lequel elle est autorisée à occuper le processeur. Lorsque son quota est écoulé, le processeur lui est retiré au bénéfice d'une autre. Si une tâche est suspendue avant l'expiration de son quantum, le scheduler en relance directement une autre.

Le choix du quantum est une affaire de compromis: si il est court alors le nombre de tâches commutées par unité de temps augmente mais l'efficacité du noyau diminue. Celle-ci peut être calculée comme suit:

$\varepsilon = (T - t) / T$ T= QUANTUM
t= TEMPS DE COMMUTATION.

Pour que ε soit le plus proche de 1, on ne peut agir que sur le quantum puisque le temps de commutation est fixé par la vitesse du processeur et la conception du scheduler.

Exemple d'efficacité (soit t=1 ms):

$\varepsilon = 0.8$ T=5 ms
$\varepsilon = 0.98$ T=50 ms

[7] Remarquons que ces trois éléments sont tous des registres.
[8] Nous sommes dans le cas d'un système avec réquisition du processeur.

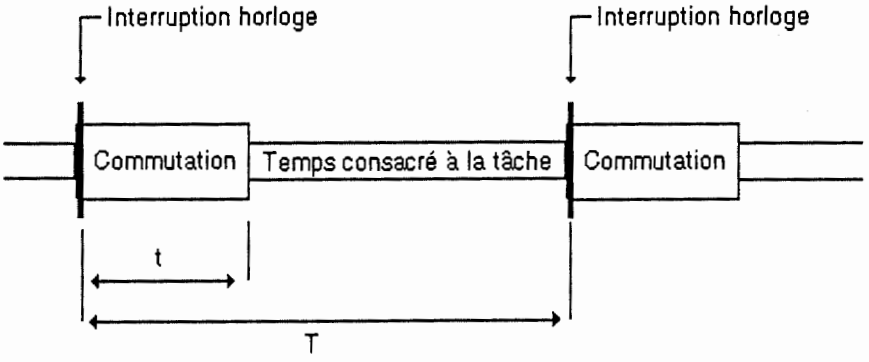


Figure 11: Utilisation du processeur durant un quantum.

Interprétons les deux efficacités de l'exemple précédent et supposons une file d'attente contenant 20 tâches dans l'état prêt. Dans le 1er cas ($\varepsilon = 0.8$), chaque tâche reprend le processeur toutes les 100 millisecondes mais nécessite presque 20% de temps en plus pour se terminer. Dans le deuxième cas, chaque tâche n'est reprise qu'une fois toutes les secondes mais se termine plus vite. Le choix du quantum est donc un compromis entre **efficacité** et **débit.**

Le scheduler, comme toutes les primitives du noyau, est caractérisé par son **temps de latence** (**latency time**). C'est la période de temps durant laquelle le système ne sait réagir à un événement, les interruptions étant masquées. Un temps de latence trop élevé peut amener des défaillances du système. Il est souvent un cheval de bataille pour la propagande des noyaux.

II. Notions spécifiques à la concurrence.

Les concepts relatifs à la concurrence détaillés dans les deux chapitres précédents sont centrés sur un seul aspect: le partage du processeur entre les tâches. Celles-ci , au terme de ces deux chapitres, paraissent comme étant indépendantes. Il n'en est rien. En effet, il existe un **couplage** entre les tâches. Ce couplage existe à deux niveaux: la **coopération** et la **compétition**. Pour éviter de faire des références en avant, je présenterai d'abord la compétition. Elle me permettra d'expliquer des concepts (les sémaphores par exemple) nécessaires à la présentation de la coopération entre tâches.

II.1 Compétition entre les tâches.

Par le fait qu'elles sont exécutées en parallèle, plusieurs tâches peuvent être en compétition. Il s'agit de l'accès aux ressources et aux périphériques du système. L'accès au processeur est résolu par le scheduler détaillé au chapitre I. Analysons maintenant les problèmes posés par l'accès aux autres ressources partagées.

II.1.A L'accès aux ressources partagées.

Dans un programme monotâche, **l'accès** aux données est **séquentiel** et c'est la structure du programme qui impose l'ordre dans lequel il s'effectue. Par contre, dans un système multitâche, l'accès aux données dépend de **l'ordonnancement** des tâches. Examinons le problème créé par cet accès:

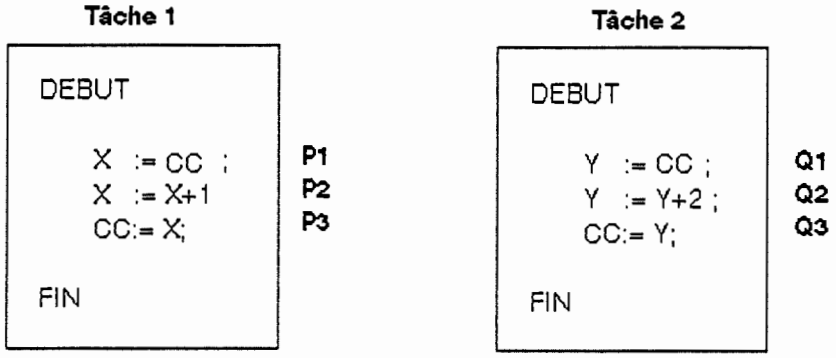


Figure 12: Accès à un objet partagé.

Selon l'ordonnancement des tâches, le résultat peut être corrompu: examinons les séquences d'exécution suivantes:

P1 P2 P3 ..... Q1 Q2 Q3 ==> résultat: CC:=CC+3
P1 ... Q1 Q2 Q3 .. P2 P3 ==> résultat: CC:=CC+1

Ce problème provient de **l'accès concurrent**, *c'est-à-dire une situation dans laquelle plusieurs tâches lisent et écrivent des données partagées, et où le résultat est fonction de* ***l'ordonnancement*** *des tâches*[9].

Nous pouvons résoudre ce problème si l'accès d'un objet par une tâche interdit son accès à d'autres tâches.

II.2.B Sections critiques et exclusion mutuelle.

Les accès concurrents peuvent être évités en interdisant l'accès simultané à une ressource partagée. Le moyen empêchant l'accès à un composant (co-processeur mathématique, ...) ou à un périphérique (imprimante , ...) si celui-ci est déjà utilisé par une autre tâche est appelé **l'exclusion mutuelle**. On appelle **section critique**, la partie du programme conduisant à un conflit d'accès. Le problème de l'accès concurrent peut ainsi être résolu en n'autorisant une seule tâche à exécuter la section critique. Mais cette condition est insuffisante. Voici les 4 règles régissant l'accès à une section critique:

1- Une section critique ne doit être occupée que par **une seule** tâche à la fois.

2- Aucune hypothèse ne doit être faite sur la **vitesse** relative et le nombre de tâche en exécution.

3- Aucune tâche suspendue en dehors d'une section critique ne doit en **bloquer** une autre.

4- Une tâche doit attendre un **temps fini** devant une section critique.

[9] D. TSCHIRHART, commande en temps réel.

Plusieurs méthodes peuvent être utilisées pour assurer l'exclusion mutuelle. En voici quatre:

Exclusion mutuelle par masquage des interruptions.

Comme nous l'avons vu précédemment [10], l'origine du problème de l'accès concurrent est **l'ordonnancement** des tâches. Une façon d'assurer l'exclusion mutuelle est donc d'empêcher ce dernier. Il suffit pour cela d'inhiber l'appel périodique au scheduler: le masquage des interruptions permet d'ignorer le 'tick' horloge du système et donc d'éviter son exécution. Le système se trouve alors en mode **monotâche** avec une seule tâche en exécution: celle qui exécute la section critique. Cette méthode allonge le temps de réponse aux interruptions et les contraintes de temps imposées par le processus peuvent ne plus être respectées (on peut alors ne masquer que l'interruption horloge; certains systèmes le permettent). Le masquage des interruptions ne concernant qu'un seul processeur, cette méthode n'est donc plus valable dans un système multiprocesseurs. De plus, beaucoup de processeurs n'autorisent le masquage des interruptions qu'en mode privilégié (80286,80386, 680x0 ....) . Une tentative de modification de masque par une tâche provoquerait alors une exception. Le noyau (toujours exécuté en mode privilégié) décidera alors des sanctions à prendre (destruction de la tâche, ignorance de l'instruction ...). Le masquage des interruptions est souvent utilisé par le noyau lorsqu'il met à jour les informations concernant le système (table des tâches, pointeurs .....)

Exclusion mutuelle par verrou et attente active.

Une deuxième solution est l'utilisation d'une variable appelée **verrou** dont l'accès est **indivisible** (sa taille est celle d'un registre). Voici comment utiliser cette variable comme solution logicielle au problème de l'exclusion mutuelle:

[10] Cfr. II.1.A

```
int VERROU=0;        /* variable globale */
```

Code de la tâche:

```
WHILE (VERROU=1);  /* attente active */
VERROU=1;
/* Début de la section critique */
.
.
.
/* Fin de la section critique */
VERROU=0;
.
.
```

Figure 13: Exclusion mutuelle par verrou et attente active

Supposons deux tâches susceptibles d'exécuter cette section critique. Avant d'y pénétrer, chacune teste le verrou. Si celui-ci est nul, la tâche le positionne à 1 et rentre dans la section critique. A cet instant, si la deuxième tâche désire également exécuter cette section critique, elle verra le verrou égal à 1 et sera forcée d'attendre son passage à zéro.

La méthode du verrou peut engendrer des erreurs: supposons que la tâche 1 en fin de quantum est remplacée par la 2 avant d'avoir eu le temps de positionner le verrou à 1. La tâche 2 voyant le verrou à 0, rentrera dans la section critique et positionnera le verrou à 1. Au cycle suivant, le scheduler rend le contrôle à la première tâche qui à son tour rentre dans la section critique: il y a un accès concurrent.

Pour remédier à ce problème, la solution peut être modifiée: on incrémente d'abord le verrou (appelé alors variable d'avertissement) , puis on le compare à 1. Si il est supérieur à 1, c'est qu'une tâche est déjà dans la section critique. Le verrou est alors décrémenté et l'opération recommence. Si il est égal à 1, la tâche peut accéder à la section critique.

```
int VERROU=0;        /* variable globale */
```

Code de la tâche:

```
WHILE  (++ VERROU>1) VERROU--;
/* Début de la section critique */
.
.
.
/* Fin de la section critique */
VERROU=0;
.
```

Figure 14: Exclusion mutuelle par verrou et attente active (2ème version)

Cette modification de la solution initiale ne permet plus à plusieurs tâches d'exécuter en même temps la section critique mais les tâches peuvent se **bloquer mutuellement.** Exécutons la séquence suivante:

TACHE 1: ++VERROU
TACHE 2: ++VERROU
TACHE 1: VERROU>1 (ici VERROU=2)
TACHE 2: VERROU>1 (ici VERROU=2)
TACHE 1: VERROU --
.
.
.

Après plusieurs tentatives, une tâche réussira à rentrer dans la section mais encore une fois cette solution comme toutes les autres n'est pas généralisable à plusieurs processeurs.

Pour éviter des erreurs (blocage mutuel ou présence de plusieurs tâches dans une section critique) les noyaux proposent deux primitives relatives aux verrous:

VERROUILLER (NUM_VERROU)
DEVERROUILLER (NUM_VERROU)

Leur exécution est **indivisible**: une tâche ayant commencé l'exécution d'une de ces primitives, ne perdra pas le processeur avant de l'avoir terminée. Leur fonctionnement est le suivant :

VERROUILLER (NUM_VERROU): si le verrou NUM_VERROU est ouvert , il est fermé et la tâche continue son exécution (elle rentre dans la section critique). Si il est fermé elle est suspendue et placée en queue de la file d'attente du verrou NUM_VERROU.

DEVERROUILLER (NUM_VERROU): si la file d'attente du verrou NUM_VERROU n'est pas vide, la première tâche est remise en exécution. Si la file est vide, le verrou NUM_VERROU est ouvert.

Ces deux primitives ont un avantage sur la solution précédente: elles permettent d'éliminer **l'attente active**. Une section critique sera donc placée entre ces deux primitives (avec le même numéro de verrou). Exemple:

```
.
.
VERROUILLER (5)   /* 5 est l'identifiant du verrou relatif à cette
                     section critique */
/* Début section critique */
.
.
/* Fin section critique */
DEVERROUILLER (5);
.
.
```

Exclusion mutuelle par instruction Test and Set.

La cause des erreurs engendrées par les solutions précédentes est le fait que le test du verrou et son positionnement à 1 n'est pas une opération indivisible: une tâche peut en effet être interrompue entre ces deux instructions. Les nouveaux micro-processeurs proposent une nouvelle instruction (BTS pour le 80386) permettant d'effectuer ces deux instructions d'une façon **atomique**. Par conséquent, une interruption ne sera servie qu'à la fin de cette instruction: aucune tâche ne peut donc modifier le verrou entre sa lecture et son écriture. Cette solution peut

aussi être généralisée à plusieurs processeurs puisque le bus est **verrouillé** durant cette opération interdisant alors d'autres processeurs d'accéder à la variable.

Exclusion mutuelle par sémaphores.

Les verrous permettent de résoudre un problème posé par l'exclusion mutuelle: assurer la présence d'une seule tâche au maximum dans une section critique. Il existe des sections dans lesquelles M tâches (au maximum) sont autorisées à rentrer. C'est le cas de l'allocation de M imprimantes à N tâches (N > M) : chaque tâche peut demander une imprimante, l'utilisera et finalement la restituera. Si aucune imprimante n'est libre lors de la demande d'une tâche, celle-ci sera suspendue. Elle sera réveillée lors de la restitution d'une imprimante par une autre tâche. Si M=1, le problème revient à celui de la section critique. Pour M>1, le problème est malaisé à résoudre par les verrous. Un nouvel outil est donc introduit (proposé par E.W. DIJKSTRA en 1968). Il est en fait une généralisation du verrou. Le champ caractérisant l'état de celui-ci est en effet booléen: ouvert ou fermé. Pour une **sémaphore** ce champs est un entier. Comme pour les verrous, l'accès aux sémaphores est régi par deux primitives du noyau baptisées P et V par DIJKSTRA et définies comme suit:

P (NUM_SEM): l'appel de cette primitive soustrait 1 à la valeur de la sémaphore. Si le résultat est négatif, la tâche est suspendue et insérée en queue de la file d'attente de la sémaphore NUM_SEM.

V(NUM_SEM): elle ajoute 1 à la valeur de la sémaphore. Si la valeur obtenue est plus petite ou égale à zéro (et donc qu'il existe au moins une tâche en attente de cette sémaphore) la tâche en tête de la file d'attente est réveillée.

La valeur initiale de la sémaphore dépend du problème: concernant notre gestionnaire d'imprimantes elle sera initialisée à M. Pour une section critique elle sera initialisée à 1(MUTEX est souvent utilisé pour désigner une sémaphore réalisant l'exclusion mutuelle):

```
                VAR MUTEX: SEMAPHORE  INIT 1;
TACHE:
                P(MUTEX)
                /* Section critique */
                V(MUTEX)
```

II.2.C Comportement dynamique.

L'exclusion mutuelle peut conduire à un **interblocage** sournois appelé interblocage mutuel (**DEADLOCK**). C'est une situation dans laquelle une tâche est définitivement bloquée en attente d'une ressource possédée par une autre tâche elle-même définitivement bloquée en attente d'une ressource possédée par la première[11]. Trois conditions doivent être rencontrées pour créer un interblocage:

1- Exclusion mutuelle: chaque tâche à l'usage **exclusif** des ressources critiques.

2- Non-préemption: la ressource n'est allouée qu'**après** avoir été libérée par le processus qui l'avait précédemment acquise..

3- Conservation des ressources pendant le blocage: quand un processus est bloqué, il garde **toutes** les ressources précédemment acquises.

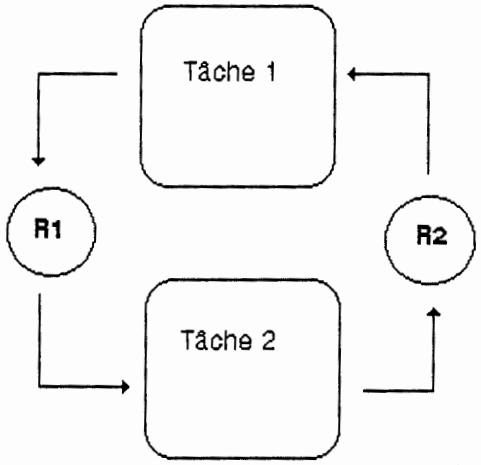


Figure 15: Exemple d'interblocage mutuel.

La tâche 1 est prête à libérer la ressource 2 (R2) si la tâche 2 libère R1 mais la tâche 2 ne libérera R1 qu'après l'obtention de R2!

[11] J. RAMAEKERS, Systèmes d'exploitation, Novembre 1990.

II.2 Coopération des tâches.

Les tâches contribuent à l'atteinte d'un objectif **partagé**. Pour atteindre cet objectif, des relations seront définies **explicitement** entre les tâches: la tâche A suspend la tâche B, la tâche C échange des informations avec la tâche D .... La coopération est présentée dans les deux sections suivantes: la **communication** et la **synchronisation**

II.2.A La communication.

La communication est l'échange d'informations entre deux tâches. Deux types de communications peuvent être utilisés:

Communication non-contrôlée

La façon la plus simple et la plus rapide de transmettre des données est de ne pas en transmettre! Il suffit pour cela de disposer d'une zone de données **commune** dans laquelle seront déposées les données en libre accès. L'allocation de cette zone est réalisée à l'aide de variables globales. L'exemple suivant écrit en PASCAL en est une illustration. La variable DONNEE_COMMUNE est une variable globale. La tâche 2 attend (elle boucle tant que DONNEE_COMMUNE est nulle) qu'une valeur (différente de zéro) soit déposée dans cette variable.

Tâche 1

```
INTEGER i;
READLN ("Solde du compte:",i);
DONNEE_COMMUNE=i;
.
.
.
.
```

Tâche 2

```
INTEGER j;
WHILE (DONNEE_COMMUNE=0);
j=DONNEE_COMMUNE;
WRITELN("Solde: ",j);
.
.
```

Figure 16: communication par variable commune.

Communication contrôlée.

L'échange d'informations est dans ce cas sécurisé au travers des services du noyau assurant ainsi implicitement **l'indivisibilité** des manipulations de données et de **l'intégrité** de l'information. Les nouveaux procédés utilisés sont ceux permettant l'échange de messages entre les tâches. Des primitives d'échange de données entre tâches sont donc proposées par le noyau:

ENVOYER_MESSAGE
RECEVOIR_MESSAGE.

Les paramètres de ces primitives diffèrent selon la cible du message.

1. La destination du message peut être une tâche.

Les primitives deviennent alors:

ENVOYER_MESSAGE (NUM_TACHE_DEST, MESSAGE)
LIRE_MESSAGE (NUM_TACHE_EXP)

Si un message en provenance de NUM_TACHE_EXP est disponible, la fonction LIRE_MESSAGE le retourne. Par contre, si aucun n'est disponible, la tâche est suspendue par le noyau. Elle sera réveillée lorsque qu'un message lui étant destiné sera envoyé par la tâche spécifiée. Le débit d'un expéditeur est aussi contrôlé par le noyau: si une tâche envoie des données plus vite que le destinataire ne les consomme, elle est également suspendue. Elle sera débloquée lorsque le consommateur sera prêt à les lire. Ce modèle de communication est celui du **producteur/consommateur.**

```
VOID PRODUCTEUR (VOID)
  {
    int DONNEE;
    DONNEE=produire()   /* production de la donnée à transmettre */
    ENVOYER_MESSAGE (num_tache(CONSOMMATEUR),&DONNEE);
  }
```

```
VOID CONSOMMATEUR (VOID)
    {
     int DONNEE;
     LIRE_MESSAGE (num_tache(CONSOMMATEUR),&DONNEE);
     consommer(DONNEE)
    }
```

Figure 17: Le modèle PRODUCTEUR/CONSOMMATEUR.

2. La destination du message peut être une boîte à lettres.

Une boîte à lettres est une **zone de mémoire** de taille fixe pouvant parfois recevoir plusieurs messages. Nos primitives de communication deviennent alors:

ENVOIE_MESSAGE (ADRESSE_DE_LA_BOITE,MESSAGE)
LIRE_MESSAGE (ADRESSE_DE_LA_BOITE)

Dans ce cas lorsqu'une tâche tente de lire une boîte à lettres vide, elle est suspendue jusqu'à la réception d'un message[12]. Le même sort est réservé à une tâche essayant d'écrire dans une boîte à lettres déjà remplie. Elle sera débloquée lorsque que la boîte à lettres concernée sera vide et donc prête à recevoir un nouveau message.

[12] Un TIMEOUT pourra être spécifié.

```
MAIN ( VOID )
  {
    char BOITE_A_LETTRES; /* Déclaration de la boîte à lettres */
    .
    .
    .
  }
```

```
VOID EXPEDITEUR ( VOID )
  {
    char MESSAGE;
    MESSAGE=produire() /* constitution du message à envoyer */
    ENVOYER_MESSAGE(&BOITE_A_LETTRE,MESSAGE);
  }
```

```
VOID LECTEUR (VOID )
  {
    char MESSAGE;
    LIRE_MESSAGE (&BOITE_A_LETTRE,MESSAGE);
    consommer(MESSAGE);
  }
```

Figure 18: Communication par boîte à lettres.

II.2.B La synchronisation.

Le processus que les tâches décrivent fixe l'organisation **relationnelle** et **temporelle** des tâches entre elles. Ces relations sont appelées **synchronisation**. Celles-ci introduisent une relation **d'ordre** entre les tâches. Ce sont en fait des mécanismes de communication où l'information échangée n'est pas consommée: elle ne sert qu'à synchroniser les tâches. On peut déceler quatre types de synchronisation:

1- Front unilatéral: tâche synchronisée par une autre tâche ou par une interruption.
2- Front bilatéral: deux tâches se synchronisent mutuellement.
3- Conjonction: tâche synchronisée par plusieurs événements.
4- Disjonction: tâche synchronisée par l'occurrence d'un événement parmi plusieurs possibles.

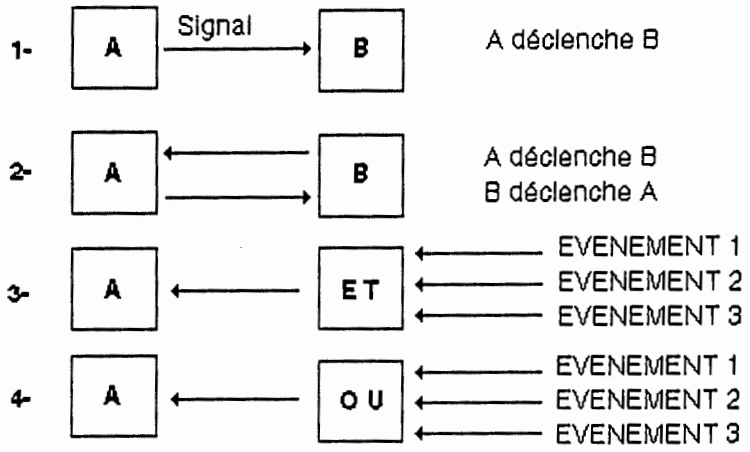


Figure 19: Les quatre types de synchronisation.

Comme pour la communication, la synchronisation peut être contrôlée ou non:

Synchronisation non-contrôlée.

L'attente active permet à une tâche de se synchroniser sur un événement en provenance d'une autre tâche. Dans l'exemple suivant, la deuxième tâche attend activement un signal en provenance de la première:

Tâche 1

```
char C;
PRINTF("Tapez ENTER pour continuer");
C=''
WHILE (C='')
   C=GETC();
SIGNAL=1;
```

Tâche 2

```
WHILE (SIGNAL=0);
/* Evénément détecté, continuer
  le processus */
.
.
.
```

Figure 20: Synchronisation par attente active.

L'attente active est un gaspillage de temps calcul: le quantum réservé à la tâche est **totalement** consommé lors de l'attente du signal. La synchronisation contrôlée permet de pallier au problème:

Synchronisation contrôlée

A part les attentes actives, les mécanismes de synchronisation relèvent des services du noyau. La synchronisation contrôlée peut être directe ou indirecte :

1. Le mécanisme de synchronisation est direct lorsque la tâche désigne directement son **destinataire**. Des primitives sont à cet effet proposées par le noyau. Elles sont du type:

ENDORMIR (NUM_TACHE)
REVEILLER (NUM_TACHE)

La fonction ENDORMIR bloque la tâche désignée. Elle ne présente pas d'intérêt dans une application temps réel où toutes les synchronisations sont connues au moment des spécifications fonctionnelles. REVEILLE permet de débloquer une tâche. Elle reprend alors son exécution à l'endroit où elle avait été interrompue.

L'exemple de la figure 21 est une illustration de l'utilisation de ces deux primitives. Il s'agit d'un spooler en attente d'un signal en provenance de l'imprimante lui signalant l'état "PRET A IMPRIMER" de celle-ci.

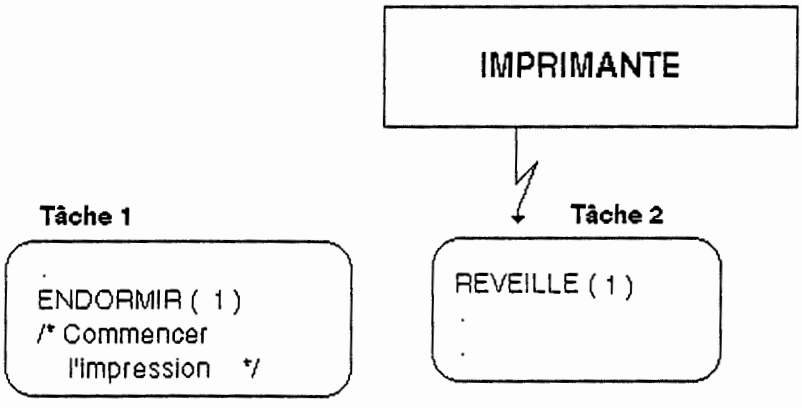


Figure 21: Synchronisation directe.

Le spooler (tâche 1) s'endort avant de commencer l'impression. Il sera réveillé par la tâche 2 chargée de la surveillance de l'imprimante. Lorsque celle-ci sera prête, la tâche 2 sera activée par interruption. Elle réveillera le spooler qui commencera l'impression. Ceci évite au spooler d'attendre activement et donc de consommer du temps calcul que d'autres taches peuvent maintenant se partager.

2. Dans le mécanisme de synchronisation indirecte, les identifiants des tâches à synchroniser ne sont pas désignés directement par l'opération. La synchronisation s'effectue par l'intermédiaire **d'objets** communs à travers des primitives du noyau assurant l'exclusion mutuelle de ces objets. Ces objets sont les variables **d'événements** et les **sémaphores.**

Les variables d'événements.

L'événement est l'outil le plus primitif permettant de résoudre le problème de synchronisation. Il peut avoir lieu ou ne pas avoir lieu. Il peut donc être représenté par une variable **booléenne** qui traduit à tout instant le fait qu'il est survenu (1) ou non (0). Deux primitives du noyau permettent de gérer les événements:

ATTENDRE (NUM_EVENT)
SIGNALE (NUM_EVENT)

La primitive ATTENDRE attend que l'événement spécifié survienne. Si celui-ci est survenu lors de son appel, la tâche continue son exécution. Par contre si il n'est pas survenu, elle est suspendue jusqu'à son apparition. L'avantage de ce type de synchronisation contrôlée est que si plusieurs tâches sont en attente d'un événement, toutes seront débloquées lors de sa survenance. Celle-ci est signalée au noyau par la primitive SIGNALE. Lorsque événement est signalé, si aucune tâche ne l'attend, deux possibilités sont à envisager: l'événement est mémorisé ou il est oublié directement. Selon D. TSCHIRHART :

"L 'utilisation des événements non mémorisés est préférable pour la commande des processus industriels. Les informations lues sur les interfaces d'entrées se périment très rapidement; il est préférable de n'activer les processus qu'au moment où les événements attendus surviennent. L'utilisation des événements non mémorisés met en jeux la vitesse des processus et peut donner lieu à des aléas de fonctionnement."

Synchronisation par sémaphores.

Les sémaphores peuvent être utilisées pour **mémoriser** un événement. Soit deux tâches devant être synchronisées par un événement: on utilisera deux sémaphores: SEM1 et SEM2 initialisées à zéro. La tâche activée par l'interruption exécutera les opérations V(SEM1) et V(SEM2). Les deux tâches à synchroniser exécuteront P(SEM1) pour la première tâche et P(SEM2) pour la deuxième. Analysons plusieurs séquences d'exécution:

- Les deux tâches (1 et 2) exécutent P(s) avant que la troisième ne réalise V(s). Elles se bloquent donc toutes les deux, les sémaphores sont égales à -1.

- La troisième tâche exécute V(s) alors que les tâches 1 et 2 ne sont pas encore bloquées: tout se passe comme si l'événement était mémorisé. Les prochains P(s) ne bloqueront aucune tâche. Si V(s) est exécuté plusieurs fois, soit n fois, avant l'exécution de P(s), chaque opération incrémente la sémaphore. Les tâches disposent alors d'un **potentiel** de n activations avant de se bloquer.

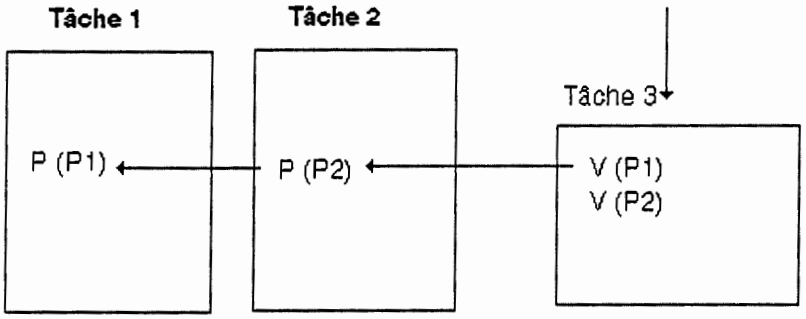


Figure 22: Synchronisation par sémaphores.

III. Les systèmes temps réel.

Avant de terminer cette première partie, je voudrais introduire les systèmes temps réel. Cette motivation me vient du fait que MYOS, le noyau développé dans la seconde partie, est conçu pour être compatible avec VRTX32, un noyau temps réel.

III.1 Définition

"Even the right answer is wrong if it is late!"

Ce slogan publicitaire de READY SYSTEMS pour la propagande de son exécutif temps réel (VRTX 32) fait apparaître une contrainte par laquelle les systèmes d'exploitation classiques ne sont pas concernés: la contrainte de **temps**. En effet:

" *un système fonctionne en temps réel lorsqu'il est capable d'absorber toutes les informations d'entrée sans qu'elles soient trop vieilles pour l'intérêt qu'elles représentent, et , par ailleurs, de réagir à celles-ci suffisamment vite pour que cette réaction ait un sens*"[13]

Retenons que des résultats **corrects** doivent être fournis à des **moments** précis. Un système temps réel doit donc satisfaire les contraintes sur le temps de traitement sans quoi des résultats tardifs ou prématurés conduisent à la défaillance du système. Ces contraintes de temps auxquelles doit satisfaire un système temps réel sont très variables selon l'application. Analysons les types de contraintes de temps après en avoir donné quelques exemples.

[13]ABRIAL - BOURGNE. Cours de temps réel, READY SYSTEMS FRANCE.

III.2 Les contraintes de temps dans un système.

Chaque tâche d'un processus est exécutée en réponse à des sollicitations externes ou internes avec des contraintes de temps fixées par le processus. Celui-ci fixe une **échéance de temps** *Te* pour chaque calcul à effectuer en réponse à des événements. C'est la présence de cette échéance qui caractérise un système temps réel: voici une échelle permettant de balayer les différents types de contraintes de temps:

RADAR	1 milliseconde
VISUALISATION	1 seconde
CONTROLE DE STOCK	30 secondes
FABRICATION	Quelques minutes
REACTIONS CHIMIQUES	Quelques heures

On peut classer ces contraintes de temps en trois catégories [14]:

III.2.A Les contraintes de temps faibles

Pour ce type de contraintes, la programmation en **mode bouclé** est autorisée. Ce mode de programmation gère les entrées-sorties par une **scrutation** permanente illustrée par la figure 23. Il est simple à mettre en oeuvre mais lors de la modification du système (l'ajout d'une entrée à lire par exemple), les contraintes de temps peuvent ne plus être respectées. Le temps d'un cycle de scrutation (Tc sur la figure 23) doit être inférieur à la contrainte de temps Te fixée par le processus. Le genre d'application pour laquelle convient bien la scrutation est par exemple la surveillance des boutons d'appel d'un ascenseur: une vingtaine de scrutations par seconde suffisent à la détection d'un appel.

[14] D. TSCHIRHART, Commande en temps réel, DUNOD,France, 1990.

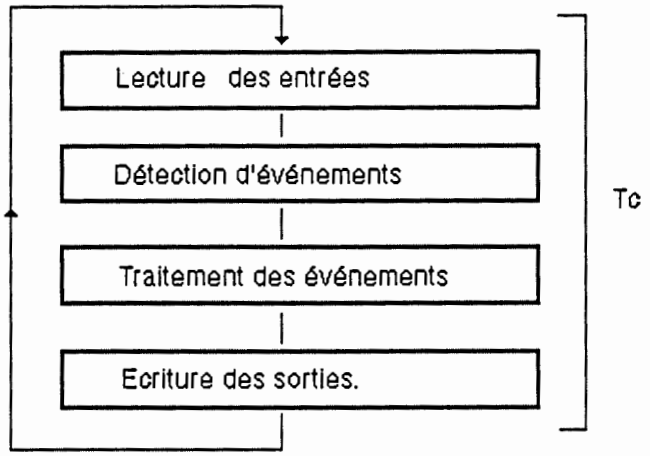


FIGURE 23: Gestion des entrées-sorties par scrutation.

III.2.B Contraintes de temps faibles avec quelques événements contraignants.

Pour certaines contraintes, il se pourrait que le temps de cycle soit insuffisant. On associe alors le traitement concerné à un mécanisme **d'interruption.** A chaque apparition d'une interruption associée à un événement, le programme en exécution sera suspendu au bénéfice du traitement urgent.

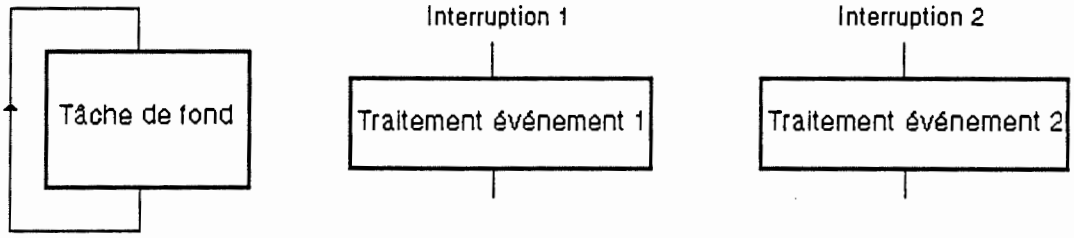


Figure 24: Gestion des entrées-sorties avec 2 événements contraignants.

Un exemple d'événement contraignant pour notre exemple précédent relatif aux ascenseurs, peut être le bouton stop dans la cabine. Un tel événement ne peut-être

détecté par scrutation, le temps de détection risquant d'être trop long (avec les conséquences que cela entraînerait: une cabine animée d'une vitesse de 5 mètres/seconde arrêtée avec 1/2 seconde de retard pourrait s'écraser sur la plate-forme supérieure.)

III.2.C Les fortes contraintes de temps.

Dans un système monoprocesseur, les tâches sont exécutées **séquentiellement**. Si les contraintes de temps sont fortes, un seul processeur peut ne plus suffire. Il faut alors ajouter des processeurs. Si les contraintes portent toujours sur les mêmes tâches, les processeurs pourront être **spécialisés** (processeurs graphiques, contrôleur d'entrée-sortie ...). Ce sont alors des processeurs **esclaves.**

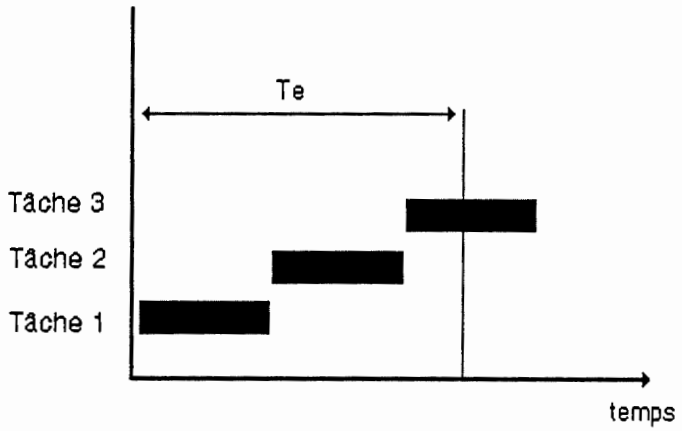


Figure 25: Exemple d'un système à contrainte insatisfaite.

2 ème Partie.

I. But de la création de MYOS.

Au laboratoire des ascenseurs SCHINDLER, le nouveau projet MICVX est destiné à remplacer les manoeuvres[1] actuelles d'ascenseurs. Actuellement celles-ci reposent sur un processeur 6809 (MOTOROLLA) programmé en monotâche. La base du programme est la scrutation permanente des entrées/sorties. A chaque événement (changement d'état d'une entrée) correspond une action (mise en route du moteur, ouverture des portes ...) Cet ancien système ne suffit plus aux nouveaux besoins: augmentation du nombre d'étages, batterie d'ascenseurs de plus en plus grande (jusqu'à seize ascenseurs travaillent en parallèle) avec optimisation des courses ...L'idée du projet MICVX est de décentraliser la manoeuvre: une partie située dans la gaine servira à la localisation de la cabine, une autre dans la cabine gérera les appels, une troisième dans la salle des machines coordonnera le tout et transmettra les décisions au module "force motrice" pour déplacer la cabine:

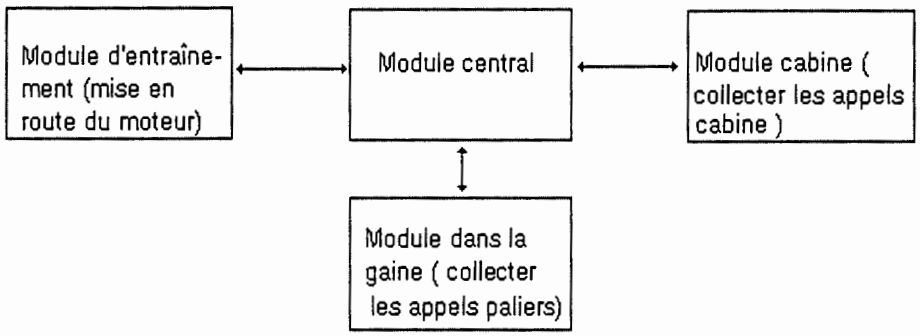


Figure 26: éclatement de la manoeuvre.

La coordination de ce véritable réseau est impensable en programmation séquentielle. Venu donc le moment aux informaticiens de la compagnie de changer de langage de programmation (le C remplace l'assembleur 6809), de cible (nouveau processeur, nouvelle carte CPU ...) et de style de programmation (programmation concurrente). L'apprentissage du C a été facilité par la compatibilité de ce langage: sa pratique a été possible sur les PC du laboratoire.

[1] On entend par manoeuvre le système électronique gérant tout le fonctionnement de l'ascenseur .

Le noyau temps réel choisi pour le projet MICVX est VRTX32 (Versatile Real Time Executive) , le dernier né de READY SYSTEMS. La version achetée par SCHINDLER est celle exécutable sur le processeur 68000 de MOTOROLLA. L'apprentissage de VRTX32 n'est donc pas possible sur PC (la version de VRTX32 pour 80386 coûte la bagatelle de 1 million de francs.) Le but de MYOS est de fournir les mêmes services que VRTX32 sur un PC-compatible. L'initiation à la programmation concurrente est donc facilitée: MYOS est compatible avec VRTX32 et est exécutable sur n'importe quel PC-compatible. Les programmes de manoeuvre d'ascenseurs pourront donc être écrits (en C) et puis testés en multitâche (grâce à MYOS) sur PC. Le résultat final pourra ensuite être facilement implémenté sur la cible (68000) puisqu'une seule compilation est nécessaire: celle générant le code pour le MOTOROLLA 68000.

Avant de réaliser chaque service de MYOS, je voudrais faire la distinction entre un noyau et un système d'exploitation:

1- un système d'exploitation offre un support logiciel complet tandis qu'un noyau est destiné à être utilisé soit sans système d'exploitation, soit comme base d'un système d'exploitation.

2- Dans un système d'exploitation les tâches sont des programmes complets (compilateurs, éditeurs ...) et portent le nom de processus. Dans un noyau elles sont des procédures d'un même programme.

3- Les noyaux sont souvent fournis dans des mémoires mortes à insérer sur la carte mère d'un système ou sous forme d'une bibliothèque de fonctions.

MYOS est un noyau multitâche (avec réquisition du processeur) livré sous forme d'une bibliothèque de fonctions écrites en langage C. Tel qu'il est conçu actuellement, il n'est pas temps réel. Mon intention était d'abord de créer un noyau correct et puis de l'optimiser en diminuant ses temps de réponses. Cette optimisation sera faite très prochainement.

II. Le noyau minimal.

Le noyau minimal est l'ensemble des services relatifs à la gestion des tâches. Nous y retrouverons évidement le scheduler mais aussi les fonctions de création, de destruction et de "modification" de tâche (changement de priorité, suspendre une tâche et réveiller une tâche.) Ces primitives travaillent sur une entité commune: la tâche. Au sein du noyau, celle-ci existe par son TCB (Task Control Bloc). Avant d'analyser cette structure de données et les primitives du noyau minimal, étudions les états d'une tâche sous MYOS.

II.1 Les tâches

II.1.A Etats d'une tâche sous MYOS.

Au départ , toutes les tâches existent lors de l'initialisation du noyau. Nous appellerons cet état l'état dormant: l'identifiant et le stack des tâches sont réservés. Dans cet état, la tâche est connue du système mais est ignorée du scheduler: la seule transition à partir de cet état est l'associer à une adresse de départ et à une priorité. **Cette transition, sous MYOS, est la création d'une tâche**. A partir de cet état créé, la tâche est dans l'état prêt: elle est candidate au processeur. A tout instant elle peut être élue par le scheduler en fonction de sa priorité: elle est alors en exécution[2]. Lorsqu'une tâche sera terminée, elle sera remise dans l'état dormant.[3] Il ne reste plus qu'un état: celui dans lequel la tâche est suspendue. Cette transition survient lorsqu'une condition de blocage apparaît. La tâche est alors ignorée du scheduler. Elle sera remise dans l'état prêt lors de la disparition de la condition de blocage.

[2] Je rappelle qu'il n'y a jamais qu'une tâche à la fois en exécution dans un système monoprocesseur.
[3] Notons que ce passage à l'état dormant peut aussi être causé par l'appel du service de destruction de tâche.

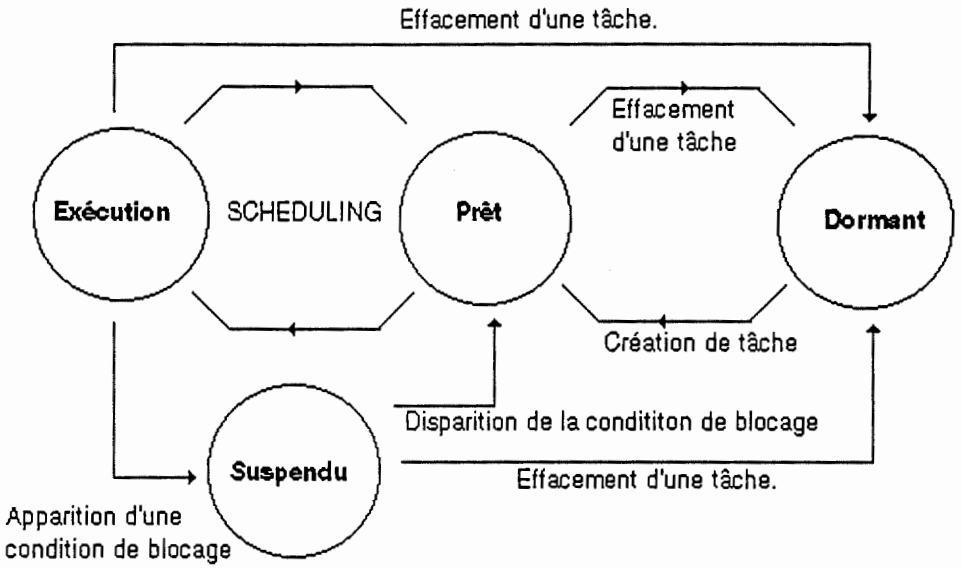


Figure 27: états et transitions entre états sous MYOS.

II.1.B Le stack

Rappelons que la pile évolue de bas en haut lors du dépôt d'une donnée et de haut en bas lors de sa récupération.

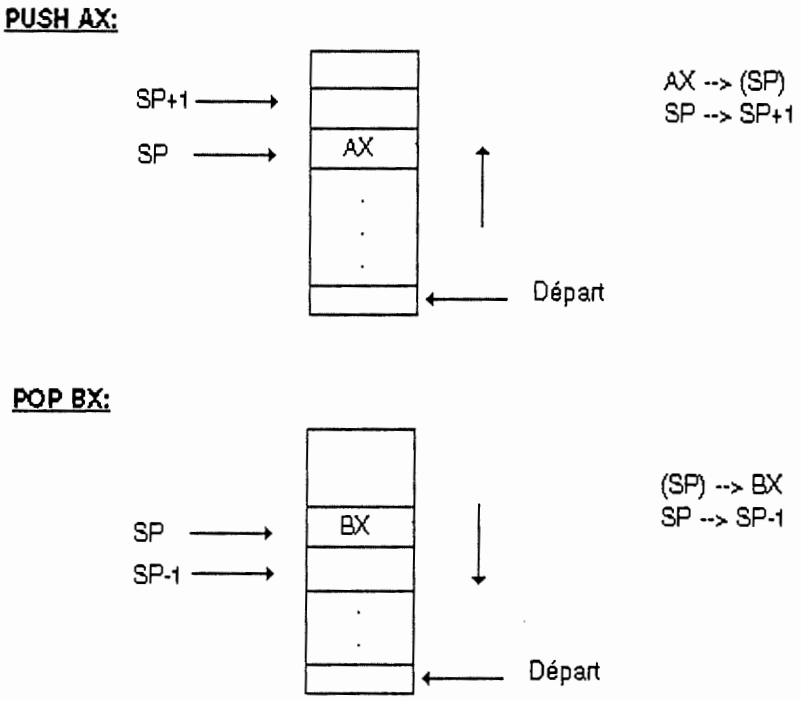


Figure 28: évolution de la pile.

Dans un système multitâche, une seule pile ne suffit plus: les tâches risquent alors de mélanger leurs données suivant leur ordonnancement: imaginons une tâche exécutant l'instruction PUSH AX en fin de quantum. Après cette instruction, c'est une autre tâche que le processeur exécute. Celle-ci pourrait exécuter POP BX "croyant" récupérer une de ses données. Réellement elle récupérera la donnée de la tâche précédente. Imaginez le désastre!

Une pile privée à chaque tâche est donc nécessaire. Pour simplifier le système, la taille maximale de la pile d'une tâche sera la même pour chaque tâche.

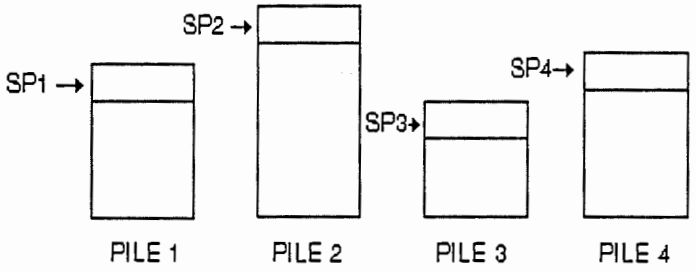


Figure 29: Modèle conceptuel des piles privées.

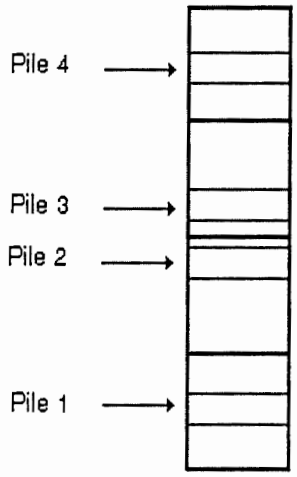


Figure 30: Modèle réel des piles privées.

La taille de la pile est un paramètre du noyau: elle pourra être ajustée selon l'application (des applications récursives par exemple).

Remarquons qu'un débordement de pile est catastrophique: il détruit la pile d'une autre tâche.

II.1.C Le TCB.

Le TCB est le descripteur de tâche. C'est une structure de données contenant toutes les informations au sujet d'une tâche. Voici sa composition sous MYOS.

id: ce champ identifie la tâche. L'identifiant est un nombre compris entre 1 et MAX_TACHE (constante définissant le nombre maximum de tâche).Il permettra de désigner une tâche lors de l'appel d'une fonction du noyau.

prio: la priorité de la tâche. Sous MYOS, la priorité d'une tâche est le nombre de tranches de temps qu'elle obtiendra à chaque fois qu'elle sera en exécution.

nb_tranche_deja_obtenue: ce champ est mis à jour par le scheduler: il indique le nombre de tranches de temps que la tâche a déjà obtenu depuis sa dernière exécution. C'est en comparant ce champ à la priorité de la tâche que le scheduler décide de la remettre dans l'état prêt ou de lui octroyer une tranche de temps supplémentaire.

statut: le statut de la tâche : dormante, créée, suspendue,en exécution.

suspension: c'est un masque représentant la ou les causes de suspension de la tâche.

stack: pointeur vers le stack de la tâche. Il est transféré dans le registre SP pour que la tâche reprenne son exécution dans le même environnement.

II.2 Primitives du noyau minimal.

II.2.A Création d'une tâche: SC-TCREATE.

Ce service crée une tâche avec un niveau de priorité, un identificateur et une adresse de départ. La tâche démarrera son exécution avec toutes les interruptions non masquées.

Cette primitive remplit les champs ID, PRIO d'un TCB de libre avec les paramètres fournis lors de l'appel. Elle initialisera aussi le pointeur de pile (STACK).

Dans un système préemptif, cet appel peut causer une commutation de tâche si la tâche nouvellement créée est plus prioritaire que la tâche de qui émane la requête.

Format de la requête:

SC-TCREATE (* ADRESSE_DEPART, INT ID, INT PRIO, INT *ERREUR)

Code de retour:

$0000 RET_OK	: pas d'erreur, la tâche est créée.
$0001 ET_TID	: erreur d'identification de tâche. L'ID est déjà utilisé.
$0002 ET_TCB	: plus de TCB de libre.

II.2.B Destruction de tâche: SC-TDELETE

Cet appel tue une tâche, éventuellement celle de qui émane la demande. La tâche tuée est remise dans l'état dormant. Une commutation de tâche a lieu si la tâche se suicide. Spécifier 0 comme identifiant de tâche provoque la destruction de la tâche elle-même.

Format:

SC_TDELETE (INT ID, INT *ERREUR)

Code de retour:

$0000 RET_OK	: pas d'erreur, la tâche est détruite.
$0001 ER_TID	: la tâche ID n'existe pas.

II.2.C Suspension de tâche: SC-TSUSPEND.

Cette fonction suspend une tâche. Lorsqu'une tâche est suspendue, une flag du TCB indique la raison de la suspension. Quand SC_TSUSPEND suspend une tâche, celle-ci ne reprendra son exécution que lorsque SC_TRESUME sera appelé. Une commutation de tâche a lieu si la tâche se suspend elle-même. Spécifier 0 comme identifiant suspend la tâche de qui émane la requête.

Format:

SC_TSUSPEND (INT ID, INT *ERREUR);

Code de retour:

$0000 RET_OK	: pas d'erreur, la tâche ID est suspendue.
$0001 ER_TID	: la tâche ID , n'existe pas

II.2.D Reprise d'une tâche (réveil): SC-TRESUME.

Ce service permet de reprendre une tâche précédemment suspendue par SC_TSUSPEND.

Format:

SC_TRESUME (INT ID, INT *ERREUR);

Code de retour:

$0000 RET_OK	: pas d'erreur, l'effet de SC_TSUSPEND est annulé.
$0001 ER_TID	: la tâche ID n'existe pas.

II.2.E Changement de la priorité: SC-TPRIORITY.

Cette fonction permet d'affecter la priorité d'une tâche. Dans un système préemptif cet appel peut causer une commutation de tâche si la nouvelle priorité de la tâche affectée est supérieure à la priorité de la tâche de qui émane la requête.

Une tâche suspendue explicitement par SC_TSUSPEND reste suspendue même si sa priorité est la plus élevée.

Spécifier 0 comme ID change la priorité de qui provient l'appel.

Format:

SC_TPRIORITY (INT ID, INT PRIO, INT *ERREUR)

Code de retour:

$0000 RET_OK	: pas d'erreur, la priorité est changée.
$0001 ER_TID	: la tâche ID n'existe pas.

II.2.F Blocage et déblocage du scheduler: SC-LOCK et SC-UNLOCK.

Cet appel (sans paramètre) bloque le scheduler, empêchant donc la commutation de tâche. La tâche garde alors le contrôle du processeur même si, dans un système préemptif, une tâche de priorité plus élevée existe.

IMPORTANT: MYOS met à jour un compteur interne de LOCK et de UNLOCK de telle manière qu'un SC_UNLOCK annule l'effet du dernier SC_LOCK. Ceci permet d'éviter la reprise prématurée de l'ordonnancement lors de l'exécution d'une section critique.

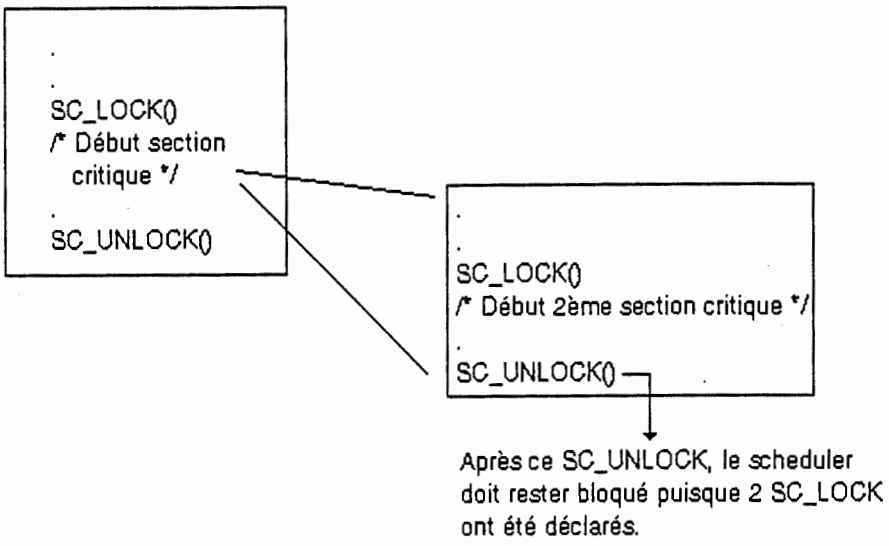


Figure 31: gestion de l'imbrication de LOCK et de UNLOCK.

Format:

SC_LOCK();

Code de retour: aucun.

II.3 L'environnement de conception de MYOS.

II.3.A Le PC-compatible.

Je rappellerai ici les particularités du PC dont il faudra tenir compte lors de la conception du noyau.

Les interruptions.

Le PC possède en début de mémoire (adresse 0000:0000) la table des vecteurs d'interruptions. Chaque vecteur contient l'adresse d'une fonction du BIOS ou du DOS. Ces fonctions peuvent donc être exécutées sans connaître leur adresse de départ (qui change d'un PC à un autre et d'un DOS à l'autre) : il suffit de poser une convention attribuant à chaque fonction un numéro. Pour demander le service d'une fonction il suffit de provoquer une interruption logicielle:

```
INT 18 /* demande le service de la fonction 18 */
```

Voici pour information les dix premières interruptions du PC:

0 CPU: Division par zéro
1 CPU: pas à pas.
2 CPU: NMI (Non Maskable Interrupt)
3 CPU: Break Point atteint
4 CPU: Débordement numérique.
5 BIOS: Copie d'écran
6 CPU: Instruction inconnue.
7 réservé.
8 HARD: IRQ0: Timer (18,2) /* IRQ= Interrupt ReQuest */
9 HARD: IRQ1: interruption du clavier.

Les deux interruptions qui nous seront utiles sont hardware: celle de l'horloge (INT 8) et celle du clavier (INT 9).

L'adressage.

Le bus d'adresse du prédécesseur du 80386, le 8088 , n'avait qu'une largeur de 16 bits, ne permettant ainsi d'adresser que 65536 (64 K) adresses. Pour étendre la capacité d'adressage à 1 million de cellules (capacité mémoire maximale d'un PC-XT) il faut 20 bits! La technique de l'époque ne permettait pas de concevoir des processeurs d'une telle complexité. Pour pouvoir contourner cette difficulté (adresser 1 méga) , il a fallu trouver une astuce: le processeur 8088 ne dispose plus d'un registre d'adresses spécial car il forme l'adresse à partir de deux registres de 16 bits qu'il combine pour former un nombre de 20 bits. Le premier d'entre eux est l'un des quatre registres de segment, le deuxième étant un autre registre ou d'une cellule mémoire. Les adresses ne sont pas simplement additionnées (ce qui ne donnerait pas une adresse de 20 bits mais tout au plus 17 bits!). En fait le contenu du registre de segment est d'abord décalé de 4 bits sur la gauche. La seconde adresse est alors additionnée à l'adresse d'une largeur de 20 bits ainsi obtenue. Ces deux adresses sont désignées sous les termes d'adresse d'offset et d'adresse de segment. L'adresse d'offset fournit le numéro de la cellule à l'intérieur d'un segment dont le début est déterminé par l'adresse de segment. L'adresse calculée par combinaison

de l'offset et du segment est appelée adresse physique car elle désigne précisément l'adresse de la cellule mémoire. Par opposition, les adresses de segment et d'offset sont appelées adresses logiques.

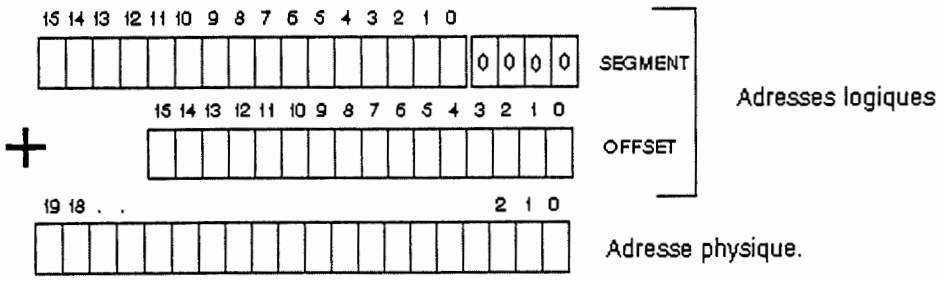


Figure 32: calcul de l'adresse physique.

Ce principe de combinaison d'adresses logiques a débouché sur une notation particulière pour indiquer l'adresse d'une cellule: on indique d'abord l'adresse du segment puis, séparée par un double point, l'adresse d'offset. Les adresses, par convention sont toujours écrites en hexadécimal. Exemple: 1234:000E désigne la cellule d'offset 14 dans le segment 1234h. Il est nécessaire de comprendre cette technique de combinaison pour concevoir le noyau.[4]

II.3.B Le compilateur QUICK C.

Le choix d'un compilateur n'a pas été difficile: le seul existant au laboratoire était QUICK C de MICROSOFT. Il est compatible avec la norme ANSI et ses librairies possèdent des fonctions propres aux 80x86. Pour la compréhension de la suite de ce chapitre, je voudrais attirer l'attention sur l'attribut INTERRUPT d'une fonction. En compilant une fonction ayant cet attribut, le compilateur génère en début de fonction le code de sauvegarde de tous les registres. A la fin de la fonction, ceux-ci seront dépilés (génération du code de récupération des registres.)

[4] Pour plus d'informations: La bible du PC, programmation système, page 2-19.

Exemple:

```
void _interrupt FONCTION_BIDON
    {
        char A;
        A=' ';
    }
```

Le code généré pour cette fonction est:

```
PUSH AX
PUSH CX
PUSH DX
PUSH BX
PUSH SP
PUSH BP
PUSH SI
PUSH DI
PUSH DS
PUSH ES

BP <-- SP
SP <-- SP-1
(BP) <-- ' '
SP <--BP

POP ES
POP DS
POP DI
POP SI
POP BP
POP SP
POP BX
POP DX
POP CX
POP AX
```

Ceci nous sera d'une grande utilité pour la conception du scheduler.

Quick C permet aussi l'insertion de code source assembleur dans le code C. Ceci est possible grâce à la directive _ASM. Exemple:

```
void fonction_bidon
    {
        int A;
        _ASM POP AX
        _ASM MOV AX,A /* on peut utiliser les
                         variables */
        .
        .
    }
```

Je voudrais enfin mentionner deux fonctions C[5]: DOS_GETVECT et DOS_SETVECT. Elles permettent respectivement de lire et de changer un vecteur d'interruption. Exemple:

```
void fonction_bidon
    {
        void *adresse;
        ADRESSE = _DOS_GETVECT (0x8);
        _DOS_SETVECT (0x9,adresse);
    }
```

Ce programme attribue à l'interruption 9, le même ISR (interrupt service routine) que pour l'interruption 8.

[5] Elles se trouvent dans la librairie DOS.

II.4 La structure de données.

Pour rester compatible avec VRTX32, nous utiliserons les mêmes constantes.

II.4.A Les constantes.

Configuration du noyau.

Trois constantes servent à la configuration du noyau: le nombre de tâches, la taille maximale de chaque pile et la taille de la pile dont l'utilité est de sauver l'environnement avant le lancement du scheduler:

```
#define   MAX_TACHE          10
#define   STACK_ALLOCATION   512
#define   SYSTEM_STACK       512
```

Codes d'erreurs:

```
#define   RET_OK     0x0000
#define   ER_TID     0x0001
#define   ER_TCB     0x0002
```

Raison de suspension:

```
#define   READY      0x0000
#define   EX_SUS     0x0001
```

II.4.B Structure des données.

Les énumérations.

Le statut du scheduler et celui des tâches sont définis sous forme d'énumération:

```
enum statut_du_scheduler
    {
        en_attente,
        en_execution
    };
enum statut_tache
    {
        dormante,
        prete,
        execution,
        suspendue
    };
```

Les structures (record).

Un descripteur de tâche (TCB) est défini comme suit:

```
struct description_tache
    {
        int id;
        int priorite;
        int nb_tranche_deja_obtenue;
        enum statut_tache statut;
        unsigned suspension;
        void *stack;
    };
```

Les variables.

L'ensemble des TCB est un tableau TACHE défini comme suit:

```
struct description_tache     tache;
```

Les variables globales sont:

```
int
    tache_en_cours  /* la tâche en exécution */
    derniere_tache_rendue_prete;
    nb_tache /* nombre de tâches créées */
    nb_tache_prete /* nombre de tâches prêtes */
    dans_section_critique /*Compteur
                            d'imbrication de
                          Sc_lock */
    reschedule /* temoin d'appel manuel au
                  scheduler */

void *debut_du_stack /* adresse de départ du
                        stack */
char *sauvetage_du_stack[2];
```

II.5 Le scheduler.

II.5.A L'exécution du scheduler.

Le scheduler est le coeur du noyau. Il commute de tâche en tâche au rythme de l'horloge du système (18,2 fois par seconde.) Sa particularité est que son exécution, contrairement à toutes les autres fonctions du noyau, n'est pas lancée à partir d'un programme (pas de CALL ni de JUMP): elle est automatique grâce au mécanisme d'interruption. Sur le PC, l'interruption de l'horloge est l'interruption N°8. Initialement, elle sert au calcul de l'heure et à

temporiser la rotation du moteur des lecteurs de disquettes. Pour dérouter cette interruption vers le scheduler, il faut remplacer dans la table des vecteurs d'interruption l'adresse du 8ème vecteur (celui de l'horloge) par l'adresse du scheduler. A ce moment, le scheduler est exécuté au rythme de +/- 18 fois par seconde. Avant d'écrire dans ce vecteur, il ne faudra pas oublier de sauver l'adresse qui s'y trouvait, d'une part pour revenir dans la situation initiale à tout moment (lorsqu'on désirera arrêter de travailler en multitâche pour retourner au DOS par exemple) et d'autre part, pour exécuter l'ancien ISR pour continuer la gestion de l'heure et de la rotation des lecteurs de disquettes.

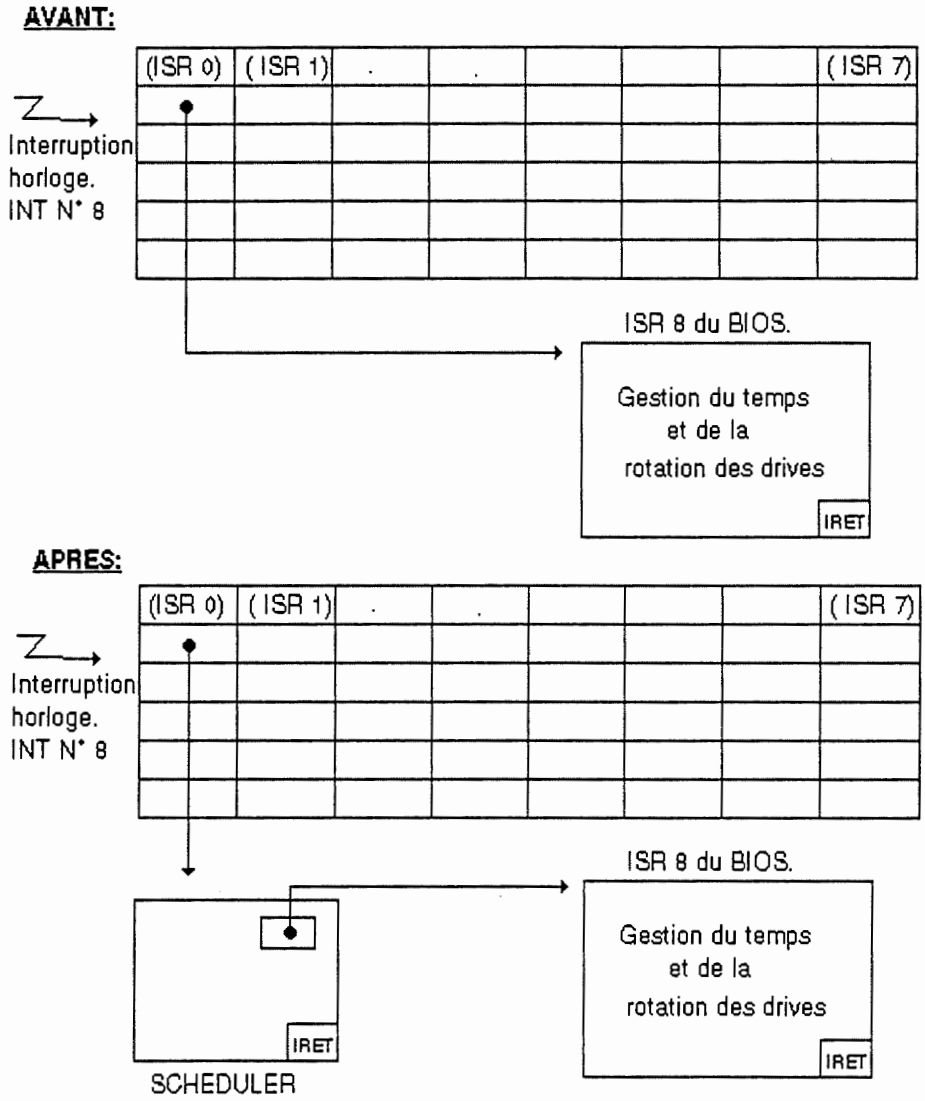


Figure 33: Détournement de l'interruption horloge vers le scheduler.

La variable qui contiendra l'ancienne adresse du vecteur 8 est OLDTIMER déclarée comme étant un pointeur vers une fonction d'interruption. A chaque exécution du scheduler (désormais automatique), il suffit d'appeler cette fonction pour assurer le calcul de l'heure et la rotation des lecteurs de disquettes.

II.5.B La commutation de tâche.

Conservation de l'environnement.

Lors de l'apparition d'une interruption, avant d'exécuter le service lui correspondant (ISR), le processeur place sur la pile trois registres: IP, le pointeur d'instruction, CS, le registre de segment, et les "flags". Ceci permet de reprendre le programme interrompu à l'endroit où il a été interrompu (IP et CS) et dans les mêmes conditions (flags).

Avoir déclaré la fonction SCHEDULER avec l'attribut _INTERRUPT nous est d'une grande utilité: tous les registres sont sauvés sur la pile avant d'exécuter le code du scheduler. Il nous suffit donc de sauver le pointeur de pile (SP) dans le TCB de la tâche interrompue. Pour redémarrer la tâche il suffira de récupérer SP dans son TCB pour pouvoir récupérer tous les registres et reprendre son exécution à l'endroit interrompu.

La commutation de tâche.

Les conditions de commutation.

Avant de commuter , il faut s'assurer qu'une commutation est permise et qu'elle a un sens. En effet, une tâche exécutant une section critique ne peut être interrompue au profit d'une autre. Pour signaler l'entrée dans une section critique, la tâche fait appel à SC_LOCK. Ce service incrémente un compteur indiquant le nombre d'imbrications de SC_LOCK. L'appel de SC_UNLOCK décrémente ce compteur appelé DANS_SECTION_CRITIQUE. Une commutation de tâche ne sera donc permise que lorsque DANS_SECTION_CRITIQUE sera nul.

Pour que la commutation ait un sens faut-il encore qu'il y ait au moins une tâche dans l'état prêt. Une deuxième condition pour commuter est: NB_TACHE_PRETE >0.

Enfin, les deux conditions précédentes étant remplies, le scheduler peut octroyer une tranche de temps supplémentaire à la tâche en fonction de sa priorité et du nombre de tranches qu'elle a déjà obtenu depuis sa dernière exécution.

La commutation.

La commutation est très simple: il suffit de changer le statut de la tâche interrompue (passage de l'état d'exécution à l'état prêt), de sauver son pointeur de pile dans son TCB et d'exécuter la tâche suivante (passage de l'état prêt à l'état d'exécution.) Pour reprendre l'exécution de la nouvelle tâche, la seule opération à effectuer est le transfert de son pointeur de pile (sauvé dans son TCB) dans SP. Le reste est automatique: la terminaison du scheduler est la récupération des registres (ceux de la nouvelle tâche puisque SP a changé)[6] et la reprise de son exécution[7].

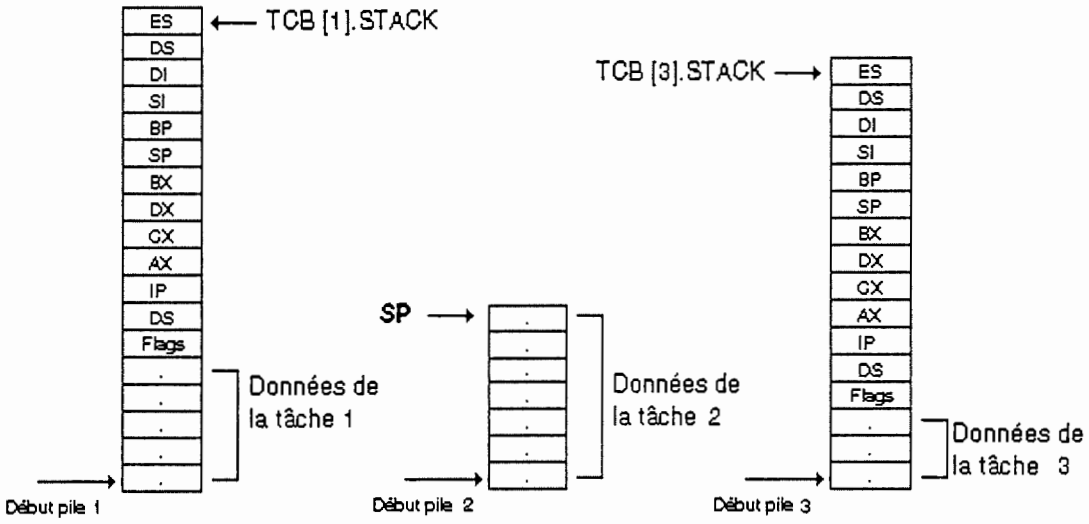


Figure 34: Avant l'apparition de l'interruption horloge la tâche 2 est en exécution

[6] Ceux de la nouvelle tâche puisque SP à changé.

[7] L'instruction de retour du scheduler generée par le compilateur est IRET (Interrupt Return) puisque qu'il s'agit d'une fonction avec l'attribut _INTERRUPT.

La figure 34 est une photo des piles lorsque la tâche 2 est en exécution (les deux autres sont dans l'état prêt.) A la fin du quantum apparaît l'interruption horloge. La figure 35 nous montre la photo des piles en rentrant dans le scheduler et après avoir sauvé tous les registres.

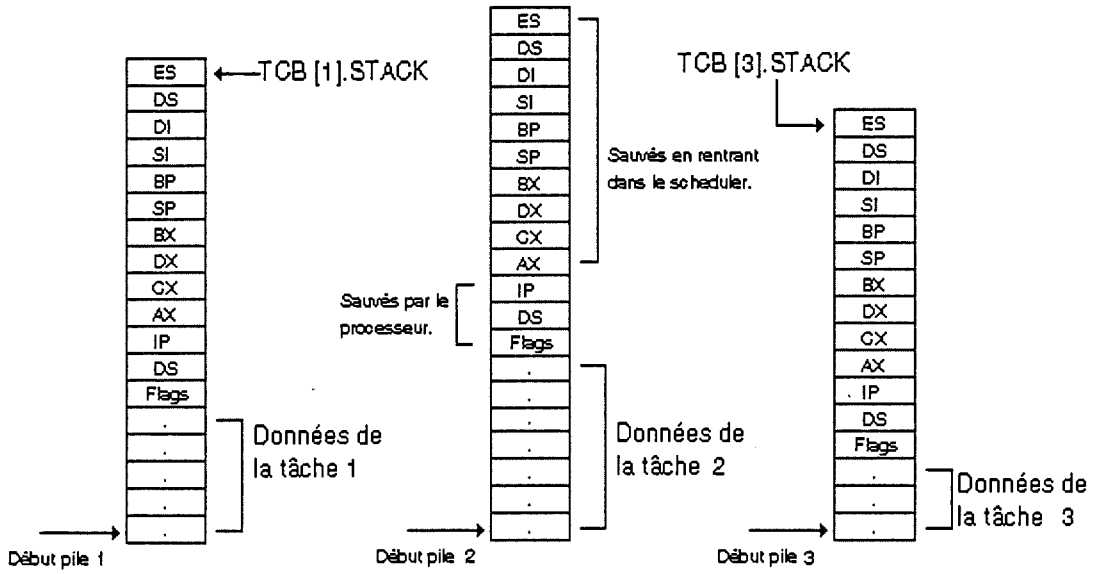


Figure 35: Apparition de l'interruption horloge: l'exécution du scheduler commence.

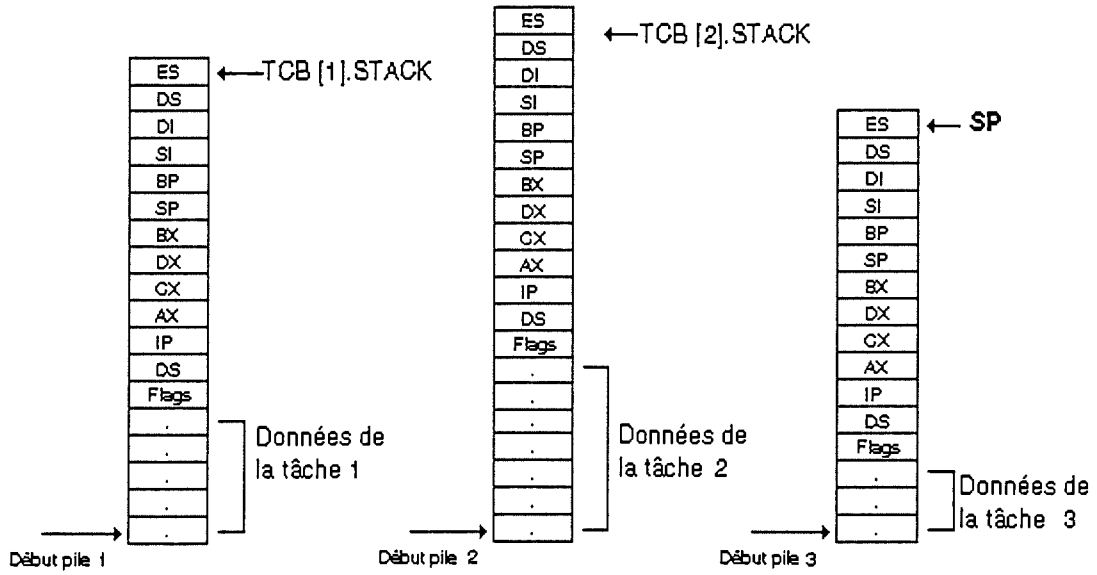


Figure 36: Choix de la prochaine tâche prête.

Une nouvelle tâche venant d'être élue par le scheduler, il ne reste plus qu'à reprendre son exécution. Cette opération est automatique: le code de terminaison du scheduler consiste en la récupération des registres et puis en l'exécution de l'instruction IRET (récupération des 3 registres sauvés par le processeur lors de l'interruption.)

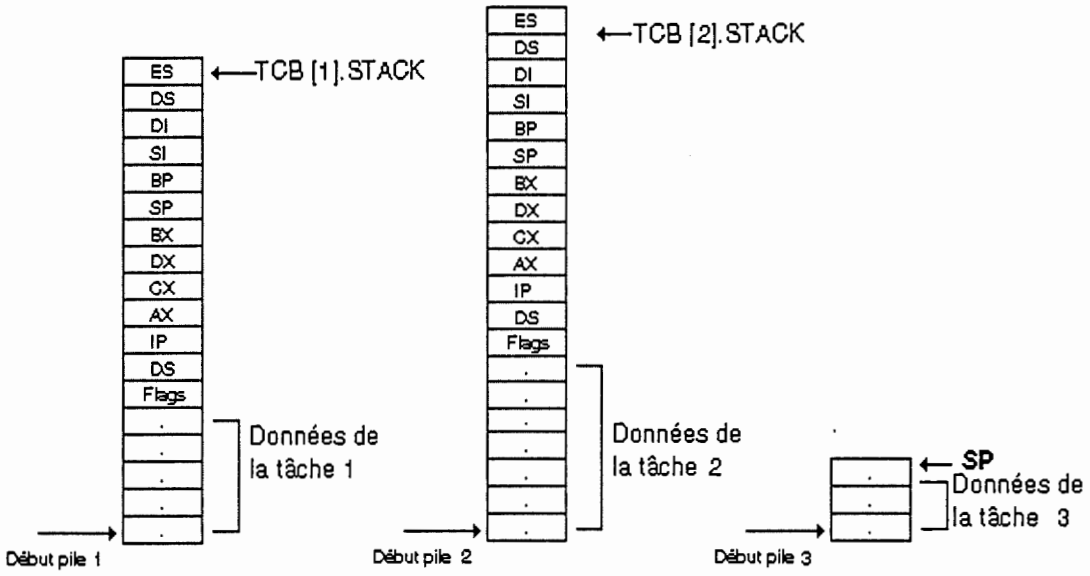


Figure 37: Reprise d'une autre tâche: la 3ème dans ce cas

Exécution "manuelle" du scheduler.

Avant d'écrire le pseudo-code du scheduler, quelques mots doivent être dits à propos de son exécution: jusqu'ici son exécution est automatique grâce au mécanisme d'interruption. Mais il existe une autre cause de son exécution: le rôle du scheduler étant de commuter , son service peut être sollicité "manuellement" par une tâche désireuse de céder à une autre le reste du quantum qui lui était réservé. C'est le cas lorsqu'une tâche se suspendra: elle appellera le scheduler[8] en lui signalant par une variable globale (RESCHEDULE) qu'il s'agit d'un appel volontaire. Une deuxième variable globale (STATUT_RESCHEDULING) indiquera au scheduler le statut qu'elle désire se donner (SUSPENDU ou DORMANT) car la raison d'un appel manuel n'est pas seulement un désir de suspension: une auto-destruction (suicide d'une tâche) peut aussi causer cet appel.

[8] Par un simple CALL.

II.5.C Le pseudo-code du scheduler.

VOID _INTERRUPT SCHEDULER (VOID)

```
TACHE[TACHE_EN_COURS].STACK <-- SP
Si(DANS_SECTION_CRITIQUE=0 et NB_TACHE_PRETE>0)
  Si(TACHE[TACHE_EN_COURS].NB_TRANCHE_DEJA_OBTENUE>=
     TACHE[TACHE_EN_COURS].PRIORITE ou RESCHEDULE=1)
    TACHE[TACHE_EN_COURS].NB_TRANCHE_DEJA_OBTENUE=0
    SI RESCHEDULE=1
      TACHE[TACHE_EN_COURS].STATUT=STATUT_RESCHEDULING
    AUSSI NON
       TACHE[TACHE_EN_COURS].STATUT=PRET
    FIN SI
    TACHE_EN_COURS=TACHE_SUIVANTE()
  FIN SI
  TACHE[TACHE_EN_COURS].STATUT=EXECUTION
  TACHE[TACHE_EN_COURS].NB_TRANCHE_DEJA_OBTENUE=
    TACHE[TACHE_EN_COURS].NB_qTRANCHE_DEJA_OBTENUE+1
FIN SI
SI RESCHEDULE=0
  EXECUTER L'ANCIEN ISR (OLDTIMER)
AUSSI NON
  RESCHEDULE=0
FIN SI
SP <-- TACHE[TACHE_EN_COURS].STACK
```

II.5.D L'idle task .

Pour développer les fonctions du noyau minimal, il me semble nécessaire de justifier la présence dans le noyau de "l'idle task". Sa fonction est d'occuper le processeur pendant "les temps creux" c'est-à-dire lorsqu'il ne sait être attribué : soit parce qu'aucune tâche n'a été créée, soit parce que toutes sont suspendues.

Sa conception est très simple: il s'agit d'une boucle dans laquelle le nombre de tâches prêtes est testé (variable NB_TACHE_PRETE). Lorsque cette variable devient positive, le contrôle est donné au scheduler qui lancera la première tâche prête qu'il trouvera.

Notons que la source de l'apparition d'une tâche prête pendant l'exécution de l'idle task ne peut être qu'un événement extérieur (la frappe d'une touche du clavier qui réveille une tâche en attente d'un caractère ou l'expiration d'un délai.) En effet, la transition à l'état prêt ne saurait pas être le résultat de l'appel d'une fonction du noyau puisqu'aucune tâche n'est en exécution.

Dans des systèmes multitâche tels que UNIX, VMS... l'idle task est appelée nul-process.

Pseudo-code de "l'idle task":

VOID IDLE_TASK (VOID)

```
STATUT_SCHEDULER=EN_ATTENTE
TANT QUE (NB_TACHE_PRETE=0)

FIN TANT QUE
STATUT_SCHEDULER=EN_EXECUTION
```

II.6 SC-TCREATE

II.6.A Conditions de création.

Sous MYOS, la création est l'association d'un TCB à une adresse de départ, une priorité et un identifiant. Pour qu'elle soit acceptée, au moins un TCB doit d'abord être libre. Ensuite son association à un ID n'est autorisée que si aucune tâche créée n'a déjà cet ID comme identifiant.

Les deux conditions ayant été remplies, les paramètres sont affectés aux champs du TCB. Pour cela, la priorité, l'adresse de départ et l'identifiant (les trois paramètres fournis à SC_TCREATE lors de son appel), sont tous les trois recopiés dans le TCB.

II.6.B La pile.

Calcul du pointeur de pile.

Contrairement à tous les autres champs du TCB, celui contenant le pointeur de pile est initialisé par le noyau. Son calcul est basé d'une part sur les deux paramètres de configuration du noyau (SYSTEM_STACK et STACK_ALLOCATION) et d'autre part sur la valeur du pointeur de pile (SP) lors du lancement du noyau. C'est une routine générée par le compilateur qui initialise SP en fonction de la longueur de la pile réservée lors de la compilation.

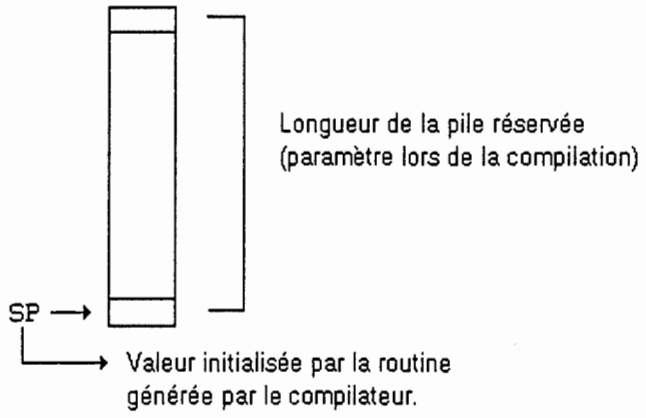


Figure 38: La pile réservée par le compilateur.

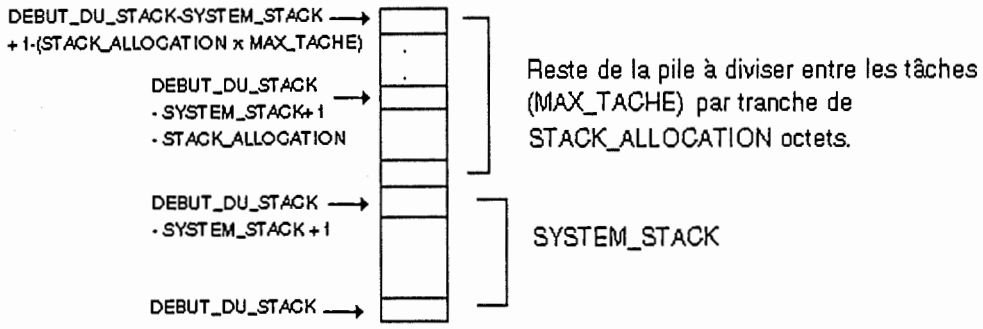


Figure 39: Division logique de la pile par le noyau.

Initialisation de la pile.

Après avoir été créée, la tâche est candidate au processeur c'est-à-dire que tôt ou tard elle sera prise en charge par le scheduler. Les premières opérations de celui-ci sont celles de récupération des registres. Il faudra donc veiller lors de la création à placer sur la pile de la tâche nouvellement créée les trois registres nécessaires au démarrage de la tâche: les flags, le pointeur d'instruction (IP) et le registre de segment (CS) ainsi que la valeur initiale de chaque registre (une valeur aléatoire est attribuée à chaque registre puisque chaque fonction générée par le compilateur initialise les registres .)

Dans un système monotâche, lorsqu'un programme est terminé, il avertit le DOS(par interruption) pour que celui-ci prenne les mesures nécessaires à la terminaison du programme (récupération de l'espace mémoire ...) Dans un système multitâche, le noyau doit aussi être averti de la terminaison d'une tâche pour libérer son TCB et la remettre dans l'état dormant. Une solution est de faire appel à la fin de chaque tâche au service de destruction de tâche pour qu'elle se suicide mais il est possible que cet appel soit automatique. En effet, à la fin de chaque procédure, le compilateur génère l'instruction RET pour rendre le contrôle au programme appelant. En plaçant au début de la pile l'adresse d'une routine du noyau, l'exécution de chaque tâche se terminera par cette routine dont le but est de mettre dans l'état dormant la tâche terminée et d'appeler le scheduler pour exécuter une autre tâche[9].

[9] Remarquons que nous sommes ici en présence d'un appel manuel du scheduler.

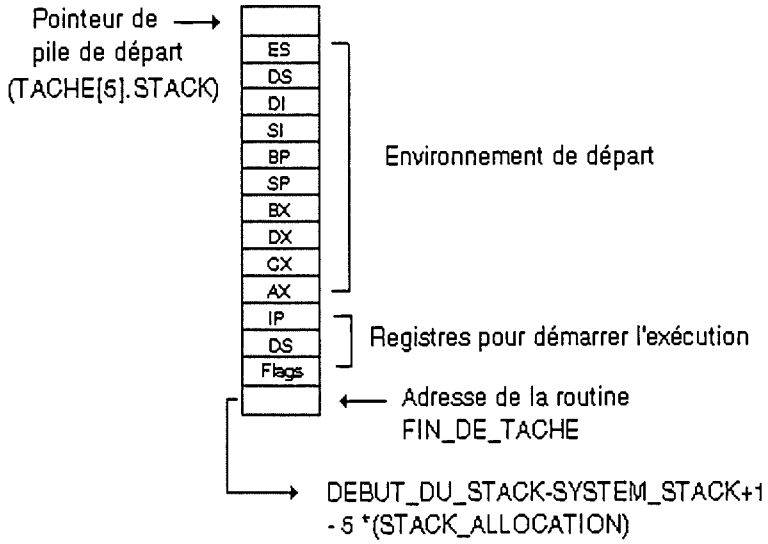


Figure 40: Garniture de la pile par le noyau lors de la création de la tâche 5.

II.6.C Pseudo-code de SC-TCREATE.

SC_TCREATE (void *ADRESSE, int ID, int PRIO, int *ERREUR)

```
SI (NB_TACHE=MAX_TACHE) *ERREUR = ER_NOCB
aussi non
  SI (TCB[ID] est libre)
    NB_TACHE=NB_TACHE+1
    TACHE[ID].STACK= DEBUT_DU_STACK -(SYSTEM_STACK *
                                      STACK_ALLOCATION)
    Placer sur la pile l'adresse de FIN_DE_TACHE
    Placer sur la pile l'adresse de départ
    Placer sur la pile la valeur initiale des registres
    TACHE[ID].statut=pret
    TACHE[ID].priorite=PRIO
    TACHE[ID].nb_tranche_deja_obtenue=0
    TACHE[ID].suspension=READY
    NB_TACHE_PRETE=NB_TACHE_PRETE+1
  aussi non
    *ERREUR=ER_TID
  fin si
fin si
```

II.6.D Pseudo-code de la routine FIN_DE_TACHE.

```
NB_TACHE=NB_TACHE-1
NB_TACHE_PRETE=NB_TACHE_PRETE-1
SI NB_TACHE_PRETE=0
  Executer IDLE_TASK
FIN SI
RESCHEDULE=1 /* appel manuel du scheduler */
STATUT_RESCHEDULING=DORMANT
Appel du scheduler
```

II.7 SC-TDELETE.

SC_TDELETE permet de tuer une tâche. Cette opération consiste à libérer le TCB de la victime. Pour cela le champ STATUT de son TCB sera affecté à la valeur DORMANT.

Le seul cas particulier à envisager est celui d'un suicide. Dans ce cas, puisque la requête de suicide émane de la tâche en exécution, le scheduler sera appelé manuellement. Il faudra également veiller à exécuter l'IDLE TASK si la seule tâche dans l'état prêt était justement celle qui se suicide.

Pseudo-code:

SC_TDELETE (int ID, int *ERREUR)

```
SI TACHE[ID] est créée
   SI ID=0
      ID=TACHE_EN_COURS
   FIN SI
   SI TACHE[ID].STATUT=SUSPENDUE
      NB_TACHE_PRETE=NB_TACHE_PRETE-1
   FIN SI
   NB_TACHE=NB_TACHE-1
   SI ID=TACHE_EN_COURS
      SI NB_TACHE_PRETE=0
         IDLE TASK
      FIN SI
      RESCHEDULE=1
      STATUT_RESCHEDULING=DORMANT
      SCHEDULER
   FIN SI
AUSSI NON
    *ERREUR=ER_TID
FIN SI
```

II.8 SC-TSUSPEND

II.8.A Les causes de suspension

Plusieurs fonctions du noyau peuvent provoquer la suspension d'une tâche: une tentative de lecture d'un caractère au clavier suspend la tâche si le buffer est vide, ainsi que la lecture d'une boîte à lettres vide, la lecture d'une sémaphore nulle ou négative ... L'effet suspensif de ces fonctions est le même: la tâche est ignorée du scheduler jusqu'à la disparition de la condition de blocage. L'appel de SC_TSUSPEND se différencie avec celui des fonctions dont l'appel peut être suspensif puisque la demande est explicite (une tâche demande la suspension d'une autre ou d'elle même) tandis que la suspension résultant des autres fonctions est implicite. C'est cette distinction qui justifie l'effet additif des causes de suspension: la levée de toutes les conditions de blocage est nécessaire au réveil de la tâche.

II.8.B La fonction SUSPEND_TACHE.

Justification de l'existence de SUSPEND_TACHE

L'effet de cette fonction est de suspendre la tâche spécifiée. Ce service est demandé par les fonctions dont le résultat peut être suspensif. Ces fonctions seront simplifiées par l'existence de SUSPEND_TACHE: elles n'auront qu'à indiquer dans le TCB de la tâche la cause de sa suspension (dans le champ SUSPENSION) avant d'appeler SUSPEND_TACHE. Lorsque la tâche sera réveillée, elle reprendra son exécution après l'instruction d'appel de SUSPEND_TACHE.

Conception de SUSPEND_TACHE.

L'opération consistant à suspendre une tâche est simple: le champ STATUT du TCB est affecté à la valeur SUSPENDU. Ensuite, si la tâche suspendue est celle en exécution, le scheduler est appelé pour reprendre l'exécution de la suivante. Mais le scheduler ne peut être appelé que si il existe au moins une tâche prête. Si ce n'est pas le cas, SUSPEND_TACHE donnera le contrôle à l'idle task. L'exécution de celle-ci se terminera lors de l'apparition d'une tâche prête. SUSPEND_TACHE continuera alors son exécution pour appeler

le scheduler sauf si la tâche rendue prête est justement celle que l'on voulait suspendre: dans ce cas SUSPEND_TACHE est terminé.

Pseudo-code de SUSPEND_TACHE:

SUSPEND_TACHE (int ID)

```
SI TACHE[ID].STATUT=PRET ou TACHE[ID].STATUT=EXECUTION
  NB_TACHE_PRETE=NB_TACHE_PRETE-1
  TACHE[ID].STATUT=SUSPENDUE
  SI (ID = TACHE_EN_COURS)
    SI (NB_TACHE_PRETE = 0)
      EXECUTER IDLE_TASK
      RESCHEDULE=1
      SI (DERNIERE_TACHE_RENDUE_PRETE=TACHE_EN_COURS)
        STATUT_RESCHEDULING=PRET
      AUSSI NON
        STATUT_RESCHEDULING=SUSPENDU
      FIN SI
      EXECUTER SCHEDULER
    AUSSI NON
      RESCHEDULE=1
      STATUT_RESCHEDULING=SUSPENDU
      EXECUTER SCHEDULER
    FIN SI
  FIN SI
FIN SI
```

II.8.C SC-TSUSPEND

La demande explicite de suspension consiste à lever dans le TCB le drapeau "suspendu explicitement" (EX_SUS) et à appeler suspend_tache.

Pseudo-code de SC_TSUSPEND:

SC_TSUSPEND (int ID, int *ERREUR)

```
SI TACHE[ID] est créée
  SI ID=0
    ID=TACHE_EN_COURS
  FIN SI
  *ERREUR=RET_OK
  Lever le drapeau EX_SUS dans TACHE[ID].suspension
  Si TACHE[ID].STATUT NOT = SUSPENDU
    SUSPEND_TACHE(ID)
  FIN SI
AUSSI NON
  *ERREUR=ER_TID
FIN SI
```

II.9 SC-TRESUME.

Cette fonction est le complément de SC_TSUSPEND. Elle permet d'annuler l'effet de ce dernier en abaissant le drapeau "suspendu explicitement" dans le TCB de la tâche concernée et si il n'existe plus de condition de blocage, la tâche sera remise dans l'état prêt.

II.9.A La fonction REPRENDRE.

Justification de l'existence de REPRENDRE.

REPRENDRE est à SC_TRESUME ce que SUSPEND_TACHE est à SC_TSUSPEND.

Reprendre (réveiller) une tâche consiste à la remettre dans l'état prêt si il n'existe plus de condition de blocage (tous les drapeaux du TCB abaissés). Comme l'effet de certaines fonctions du noyau (lire un caractère au clavier, attendre un message ...) peut être le réveil d'une tâche, REPRENDRE simplifiera leur conception: ces fonctions abaisseront le drapeau adéquat avant d'appeler REPRENDRE qui se chargera de remettre ou non la tâche dans l'état prêt.

Conception de REPRENDRE.

Puisqu'abaisser un drapeau de suspension consiste à éteindre le bit correspondant, tester si une tâche peut être remise dans l'état prêt revient à comparer la valeur du champ SUSPENSION de son TCB. Si elle est nulle (absence de raison de suspension) la tâche peut être remise dans l'état prêt.

Pseudo-code:

REPRENDRE (INT ID)

```
SI TACHE[ID].SUSPENSION = 0
   NB_TACHE_PRETE=NB_TACHE_PRETE+1
   TACHE[ID].STATUT=PRETE
   DERNIERE_TACHE_RENDUE_PRETE=ID
FIN SI
```

II.9.C SC-TRESUME

La demande explicite de reprise consiste à abaisser le drapeau "explicitement suspendu" et à solliciter le service de REPRENDRE:

Pseudo-code:

SC_TRESUME (INT ID, INT *ERREUR)

```
SI TACHE[ID] est créée
   SI ID=0
      ID=TACHE_EN_COURS
   FIN SI
   *ERREUR=RET_OK
   abaisser le drapeau EX_SUS dans TACHE[ID].suspension
   REPRENDRE (ID)
AUSSI NON
     *ERREUR=ER_TID
FIN SI
```

II.10 SC-TPRIORITY

Cette fonction est facile à réaliser. Après avoir testé l'existence de la tâche dont on souhaite changer la priorité, le champ PRIORITE de son TCB est remplacé avec la nouvelle priorité transmise comme paramètre à SC_TPRIORITY. Le noyau initial de MYOS n'étant pas temps réel, le changement de priorité n'occasionne jamais un "rescheduling".

Pseudo-code:

SC_TPRIORITY (int ID, int PRIORITE, int *ERREUR)

```
SI TACHE[ID] existe
    SI ID=0
        ID=TACHE_EN_COURS
    FIN SI
    TACHE[ID].PRIORITE=PRIORITE
    *ERREUR=RET_OK
AUSSI NON
    *ERREUR=ER_TID
FIN SI
```

II.11 SC-LOCK et SC-UNLOCK

Ces deux fonctions de gestion des sections critiques sont développées sans commentaire.

II.11.A SC-LOCK

Pseudo-code:

SC_LOCK()

```
DANS_SECTION_CRITIQUE=DANS_SECTION_CRITIQUE+1
```

II.11.B SC-UNLOCK

Pseudo-code:

SC_UNLOCK ()

```
DANS_SECTION_CRITIQUE=DANS_SECTION_CRITIQUE-1
```

II.12 Initialisation du noyau: MYOS_INIT

MYOS_INIT est le premier service du noyau à être sollicité. Il initialise les variables du noyau: pointeurs, TCB, files d'attente ..... L'appel d'un service du noyau sans avoir initialisé ce dernier peut conduire à un blocage du système ou à des résultats erronés.

Variables à initialiser:

MYOS_INIT ()

```
DANS_SECTION_CRITIQUE=0
NB_TACHE=0
NB_TACHE_PRETE=0
TACHE_EN_COURS=0
∀ tache, 0 <= TACHE[tache] <= MAX_TACHE
    TACHE[tache].statut=DORMANT.
```

II.13 Lancement de l'ordonnancement: MYOS_GO()

II.13.A Conditions de lancement.

Après avoir initialisé le noyau et créé au moins une tâche, l'appel de MYOS-GO provoque le lancement du scheduler: le système travaille alors en mode multitâche. Mais l'ordonnancement ne peut être effectué que si il existe au moins une tâche créée. La condition de lancement de l'ordonnancement est donc: NB_TACHE_PRETE>0

II.13.B Conservation de l'environnement initial.

Lors de l'arrêt du mode multitâche (signalé par MYOS_STOP), l'exécution doit continuer après l'appel de MYOS_GO. Avant l'exécution du scheduler, il est donc nécessaire de conserver tous les registres (y compris IP,CS et les flags.) Une solution est de les sauver dans une zone de données créée pour la cause. Une autre est de considérer le programme appelant MYOS_GO comme une tâche. Un TCB lui sera réservé (le TCB[0])[10]. Aucune opération de sauvetage des registres ne doit donc être pensée: le scheduler s'en charge. Pour éviter que son exécution ne continue, la tâche 0 sera mise dans l'état dormant.

Pseudo-code:

MYOS_GO()

```
RESCHEDULE= 1
STATUT_RESCHEDULING=DORMANT
Sauver l'adresse de l'ancien ISR(8)
Définir l'adresse du scheduler comme étant l'ISR(8)
SI NB_TACHE_PRETE>0
   APPELER LE SCHEDULER
FIN SI
```

[10] Notons qu'aucune fonction du noyau ne peut altérer la tâche 0 puisque l'appel d'une fonction avec l'identifiant 0 désigne la tâche de qui émane la requête.

II.14 Arrêt de l'ordonnancement: MYOS_STOP

La fin de l'ordonnancement est signalée par l'appel du service MYOS_STOP. Le scheduler est alors dissocié de l'interruption horloge qui déclenche maintenant l'ISR dont l'adresse est conservée dans OLDTIMER. Le contrôle est ensuite rendu au programme ayant lancé l'ordonnancement.

Pseudo-code:

MYOS_STOP()

```
DEFINIR OLDTIMER COMME ETANT L'ADRESSE DE L'ISR(8)
```

III. Gestion du clavier

III.1 SC-GETC

III.1.A Nécessité d'une fonction pour la gestion du clavier

En programmation séquentielle, l'attente d'un caractère est active:

```
TANT QUE (BUFFER EST VIDE)
FIN TANT QUE
/* Un caractère a été tapé */
LIRE LE PREMIER CARACTERE DU BUFFER
```

Cette solution n'est pas pensable dans un système multitâche (et encore moins dans un système temps réel): l'attente d'un caractère consomme du temps CPU. Grâce à la possibilité de suspendre une tâche, l'attente d'un caractère peut être "optimisée"[11]: lorsqu'une tâche tente de lire un caractère dans le buffer, elle sera suspendue si celui-ci est vide. C'est seulement lors de la frappe d'une touche du clavier qu'elle sera réveillée pour qu'elle puisse récupérer le caractère dans le buffer.

III.1.B Conception de SC-GETC

SC_GETC renvoie le premier caractère du buffer si celui-ci n'est pas vide. Dans le cas contraire, elle se suspend en levant le drapeau "en attente d'un caractère". C'est l'activation d'une touche, détectée par l'interruption qu'elle déclenche, qui réveillera la tâche.

[11] Elimination de l'attente active

Pseudo-code

CHAR SC_GETC (INT *ERREUR)

```
SI NB_ATTENTE_CLAVIER=1
   *ERREUR=ER_CIU /* Channel already in use */
AUSSI NON
   *ERREUR=RET_OK
   SI BUFFER EST VIDE
      NB_ATTENTE_CLAVIER=1
      ATTENTE_CARACTERE=TACHE_EN_COURS
      LEVER LE DRAPEAU CH_SUS dans le TCB[TACHE_EN_COURS]
      SUSPEND_TACHE(TACHE_EN_COURS)
      NB_ATTENTE_CLAVIER=0
      LIRE PREMIER CARACTERE DU BUFFER
      RETOURNER CE CARACTERE
   AUSSI NON
      LIRE PREMIER CARACTERE DU BUFFER
      RETOURNER CE CARACTERE
   FIN SI
FIN SI
```

III.2 NEW_KB

A elle seule, SC_GETC ne suffit pas: si aucun caractère n'est présent, son effet est suspensif mais lors du déclenchement d'une touche, la tâche en attente d'un caractère doit être réveillée. C'est NEW_KB (New Keyboard), une nouvelle procédure du noyau déclenchée par l'interruption 9 (celle du clavier) qui assurera ce réveil. Avant de dévier l'interruption 9 vers NEW_KB, nous sauverons dans OLD_KB l'adresse de l'ancien ISR. Le service de ce dernier sera sollicité par NEW_KB pour ne pas devoir gérer d'une part le remplissage du buffer et d'autre part

l'identification de la touche enfoncée [12]. Ensuite, si il en existe une, NEW_KB réveillera la tâche en attente d'un caractère.

```
APPELER OLD_KB
SI NB_ATTENTE_CLAVIER=1
   SI LE BUFFER N'EST PAS VIDE[13]
      NB_ATTENTE_CLAVIER=0
      ABAISSER LE DRAPEAU CH_SUS dans TCB[ATTENTE_CARACTERE]
      REPRENDRE (ATTENTE_CARACTERE)
   FIN SI
FIN SI
```

III.3 Affectation du noyau minimal

III.3.A MYOS_INIT

La variable NB_ATTENTE_CLAVIER devra être initialisée dans MYOS_INIT:

```
NB_ATTENTE_CLAVIER=0
```

III.3.B SC-TDELETE

Si la tâche tuée est celle en attente d'un caractère, le clavier devra être libéré :

```
SI (ATTENTE_CLAVIER=ID)
   NB_ATTENTE_CLAVIER=0
FIN SI
```

[12] Avec tous les problèmes que cela implique: les claviers AZERTY, QWERTY ...

[13] Ce n'est pas parce qu'une touche du clavier a été actionnée qu'un caractère est disponible (la frappe de la touche SCHIFT par exemple)

III.3.C MYOS_GO

Pour démarrer le mode multitâche, comme pour l'interruption de l'horloge déviée vers le scheduler, l'interruption du clavier devra être déviée vers NEW_KB

```
Définir l'adresse de NEW_KB comme étant l'ISR(9)
```

III.3.D MYOS_STOP

Pour rétablir la table des vecteurs d'interruptions dans son état initial, OLD_KB redevient l'ISR N°9:

```
Définir l'adresse OLD_KB comme étant l'ISR(9)
```

IV. La gestion du temps

IV.1 SC-DELAY

IV.1.A Nécessité d'une fonction pour la gestion du temps

Comme pour l'attente d'un caractère au clavier, la temporisation par attente active devra être abolie non seulement parce qu'elle consomme du temps CPU mais aussi par le fait que nous ne disposons plus d'un étalon de mesure. En effet, en programmation séquentielle (en supposant que le processeur exécute 1000 itérations en 1 seconde), une temporisation de 5 secondes sera implémentée par 5000 itérations:

```
FOR (I=1;I<=5000;I=I+1); /* 5000 ITERATIONS */
.
.
. Suite de la procédure
```

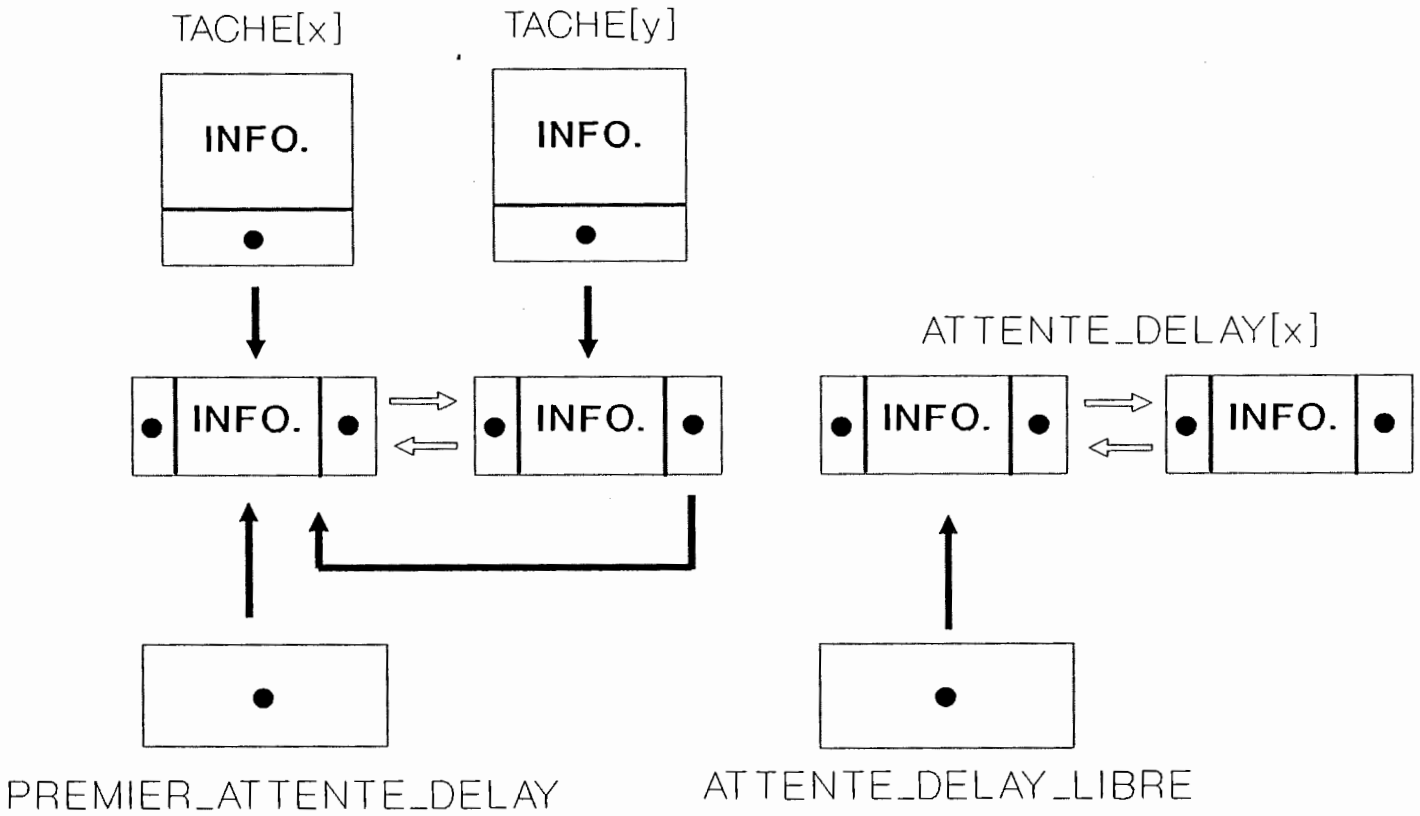
En multitâche, puisque entre deux instructions de la boucle d'autres instructions d'une autre tâche peuvent être exécutées, le temps d'exécution de 5000 itérations est fonction du nombre de tâches dans l'état prêt et de leur priorité. Une fonction du noyau doit donc être prévue pour gérer les délais.

IV.1.B Conception de SC-DELAY

SC_DELAY endort une tâche pendant le nombre de tick spécifié comme paramètre (1 tick correspond à une interruption horloge.) Une temporisation de 1 seconde sera obtenue par SC_DELAY(18).

SC_DELAY est la première fonction du noyau à introduire une nouvelle structure de données: les listes doublement chaînées:

Délais et file d'attente.



● = NULL pointer.

Cette liste contient les informations suivantes nécessaires à la gestion du temps:

ID: Identification de tâche en attente d'un délai
TIMEOUT: le nombre de tick restant avant de réveiller la tâche.
ORIGINE: =TIMEOUT ou DELAY. On signale ici si la demande d'un délai est explicite (SC_DELAY) ou implicite (spécification d'un TIMEOUT lors de l'attente d'un message par exemple)
ADRESSE_CODE_ERREUR: lorsque l'origine d'un délai est un TIMEOUT, le noyau doit renvoyer un code d'erreur dans une variable. Ce champ contient l'adresse de cette variable.

Les primitives suivantes permettent les opérations fondamentales sur la liste doublement chaînée:

AJOUTER_ATTENTE_DELAY (INT TACHE, LONG TIMEOUT, ENUM ORIGINE_DELAY ORIGINE, INT *ADRESSE_CODE_ERREUR)

Après avoir levé le drapeau "en attente d'un délai" cette primitive crée un descripteur d'attente qu'elle raccroche à la liste existante.

ENLEVER_ATTENTE_DELAY (INT TACHE)

Enlève de la liste le descripteur de la tâche spécifiée et abaisse le drapeau "en attente de délai" dans le TCB de la tâche concernée.

Grâce à ces deux primitives[14] la conception de SC_DELAY est simplifiée:

[14] Elle ne sont pas développées dans ce document. Elles consistent en une mise à jour de pointeurs..

Pseudo-code:

```
AJOUTER_ATTENTE_DELAY (TACHE_EN_COURS, TIMEOUT, DELAY)
SUSPEND_TACHE (TACHE_EN_COURS)
```

IV.2 L'écoulement du temps.

SC_DELAY ne suffit pas à la gestion des délais: elle permet d'enregistrer la requête et de suspendre la tâche envieuse de s'endormir. Après cette suspension, le noyau doit assurer le réveil de la tâche. Ce rôle est celui de SURVEILLANCE_DELAY. Son exécution est lancée à partir du SCHEDULER. Elle est donc exécutée au rythme de l'horloge. Son travail est de décrémenter le champ TIMEOUT dans chaque descripteur de la liste des attentes des délais. Si un TIMEOUT devient nul, le drapeau "en attente d'un délai" est abaissé et le service de REPRENDRE est sollicité pour réveiller la tâche concernée si il n'existe pas une autre cause de suspension.

Pseudo-code:

VOID SURVEILLANCE_DELAY (VOID)

```
SI (LA LISTE N'EST PAS VIDE)
   ∀ Eléments ATTENTE_DELAY de la liste:
        TIMEOUT=TIMEOUT-1
        SI TIMEOUT=0
             SI ORIGINE=TIMEOUT
                  *CODE_ERREUR=ER_TMO[15]
             FIN SI
             ENLEVER_ATTENTE_DELAY ( TACHE_ID )
             REPRENDRE ( TACHE_ID )
        FIN SI
FIN SI
```

[15] Puisqu'il s'agit d'une demande implicite il faudra songer ici à enlever de la file d'attente des messages (ou de sémaphore) la tâche concernée.

IV.3 Affectation du noyau minimal.

IV.3.A Les types de données.

Une structure de données est créée pour définir le type de chaque élément de la liste. Le TCB de chaque tâche possède aussi un champ supplémentaire **ATTENTE_DELAY** qui est un pointeur vers le descripteur d'attente pour avoir un accès direct à ce dernier:

```
ENUM ORIGINE_DELAY
 {
      TIMEOUT,
      DELAY
 }

STRUCT DESCRIPTION_ATTENTE_DELAY
 {
      INT  TACHE_ID
      LONG TIMEOUT
      ENUM ORIGINE_DELAY ORIGINE
      INT  *ADRESSE_CODE_ERREUR
      STRUCT DESCRIPTION_ATTENTE_DELAY *PROCHAIN_ATTENTE
      STRUCT DESCRIPTION_ATTENTE_DELAY *PRECEDENT_ATTENTE
 }
```

IV.3.B Les variables

La liste doublement chaînée est créée au travers d'un tableau:

```
STRUCT DESCRIPTION_ATTENTE_DELAY  ATTENTE_DELAY [MAX_TACHE]
```

IV.3.C SC-TCREATE

A la création de la tâche, le champ ATTENTE_DELAY devra être initialisé:

```
TACHE[ID].ATTENTE_DELAY= NULL
```

IV.3.D SC-TDELETE

Lors de la destruction d'une tâche, si la tâche victime est en attente d'un délai, son descripteur d'attente doit être ôté de la liste:

```
SI TACHE [ID].ATTENTE_DELAY NOT = NULL
    ENLEVER_ATTENTE_DELAY (ID)
FIN SI
```

IV.3.E Le SCHEDULER

L'appel de SURVEILLANCE _DELAY devra être assuré à chaque exécution du SCHEDULER:

Au début du SCHEDULER:

```
APPELER SURVEILLANCE_DELAY
```

V. Gestion de la communication

VI.1 Les boîtes à lettres.

Pour rester compatible avec VRTX32, le modèle de communication est celui de la boîte à lettres[16]. Au sein du noyau celle-ci est représentée par l'adresse d'une cellule mémoire. Le message est le contenu de cette cellule. La valeur 0 pour un message signifie que la boîte est vide. Les boîtes à lettres ne sont pas des objets du noyau, elles sont gérées par l'utilisateur. Pratiquement il s'agit d'une variable caractère.

VI.2 Les fonctions

VI.2.A Attente d'un message: SC-PEND

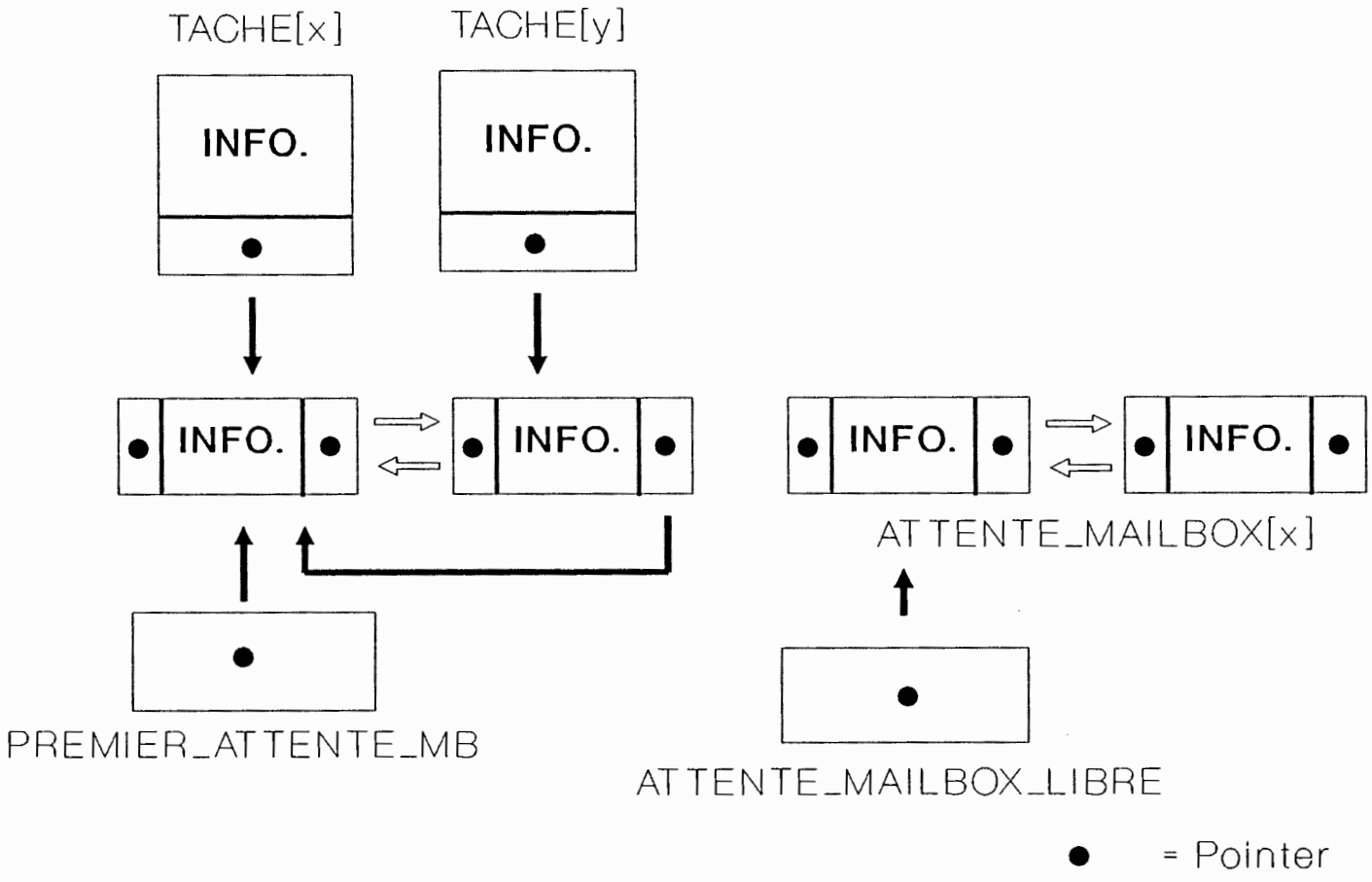
La première fonction est celle permettant l'attente d'un message d'une boîte à lettres spécifiée comme paramètre. Si lors de la requête, la boîte à lettres contient un message, il est donné au lecteur. Par contre, celui-ci est suspendu lors d'une tentative de lecture d'une boîte vide. Sa requête est alors introduite dans une file d'attente (FIFO) comportant l'identifiant de la tâche lectrice et l'adresse de la boîte à lettres.

Un TIMEOUT pourra être spécifié lors de la lecture si le message n'a plus de valeur après le délai spécifié. Un TIMEOUT de 0 signifie que la tâche attend le message jusqu'à son apparition.

Structure de la file d'attente:

[16] Par opposition au modèle producteur/consommateur.

Mailboxes et file d'attente.



Comme pour la liste des délais, deux opérations permettent d'altérer cette file d'attente:

AJOUTER_ATTENTE_MB (INT TACHE, CHAR *ADRESSE_BAL)

ENLEVER_ATTENTE_MB (INT TACHE)

Pseudo-code:

CHAR SC_PEND (char *BAL, long TIMEOUT, int *ERREUR)

```
ERREUR=RET_OK
SI *BAL NOT = 0      /* BOITE A LETTRES REMPLIE */
   RESULTAT=*BAL
   *BAL=0
AUSSI NON
   Lever le drapeau "en attente d'un message"
   AJOUTER_ATTENTE_MB (TACHE_EN_COURS, &BAL, &RESULTAT)
   SI TIMEOUT > 0
      AJOUTER_ATTENTE_DELAY (TACHE_EN_COURS, TIMEOUT,
                             TIMEOUT,&ERREUR)
   FIN SI
      SUSPEND_TACHE (TACHE_EN_COURS)
FIN SI
RETOURNER RESULTAT
```

VI.2.B Envoi d'un message: SC-POST

Ce service permet de déposer un message dans une boîte à lettres (tous les deux spécifiés comme paramètre.) Si la boîte est remplie, l'opération n'est pas autorisée et un message d'erreur est retourné. Par contre, si elle est vide, le message peut y être déposé sauf si il existe au moins une tâche en attente d'un message en provenance de cette boîte. Dans ce cas, le message lui est directement fourni sans transiter par celle-ci.

Pseudo-code:

VOID SC_POST (char *BAL, char MESSAGE, int *ERREUR)

```
SI MESSAGE=0
   *ERREUR=ER_ZMW
AUSSI NON
   SI *BAL NOT=0
      *ERREUR=ER_MIU
   AUSSI NON
     SI IL EXISTE UNE TACHE EN ATTENTE D'UN MESSAGE DANS BAL
        SOIT TACHE_ID LA PREMIERE D'ENTRE ELLE
        ENLEVER_ATTENTE_MB (TACHE_ID)
        SI TACHE[TACHE_ID].ATTENTE_DELAY NOT= NULL
           ENLEVER_ATTENTE_DELAY (TACHE_ID)
        FIN SI
        REPRENDRE (TACHE_ID)
     AUSSI NON
        *BAL=MESSAGE
     FIN SI
   FIN SI
FIN SI
```

V.3 Affectation du noyau minimal

V.3.A La structure de donnée

La description des éléments constituant la file d'attente est définie comme suit:

```
STRUCT DESCRIPTION_ATTENTE_MB
 {
      INT TACHE_ID
      CHAR *ADRESSE_BOITE_A_LETTRE
      CHAR *ADRESSE_RECEPTION
      STRUCT DESCRIPTION_ATTENTE_MB *PROCHAIN_ATTENTE_MB
      STRUCT DESCRIPTION_ATTENTE_MB *PRECEDENT_ATTENTE_MB }
```

La file d'attente existe au sein du noyau dans un tableau:

```
STRUCT DESCRIPTION_ATTENTE_MB ATTENTE_MB[MAX_TACHE]
```

Le TCB de chaque tâche contiendra en plus un pointeur vers un descripteur de la file d'attente des messages.

V.3.B SC-TDELETE.

Si la tâche victime est en attente d'un message, son descripteur dans la file d'attente des messages devra être ôté.

```
SI TACHE[TACHE_ID].ATTENTE_MB NOT = NULL
    ENLEVER_ATTENTE_MB (TACHE_ID)
FIN SI
```

V.3.C SC-TCREATE

Le champ ATTENTE_MB du TCB de la nouvelle tâche devra être initialisé:

```
TACHE[TACHE_LIBRE].ATTENTE_MB=NULL
```

V.3.D SURVEILLANCE-DELAY.

L'apparition d'un TIMEOUT nul doit provoquer le retrait du descripteur dans la file d'attente des messages si la tâche était en attente d'un message[17]:

```
SI TACHE[NUMERO_TACHE].ATTENTE_MB NOT = NULL
    ENLEVER_ATTENTE_MB (NUMERO_TACHE)
FIN SI
```

[17] En ayant donc spécifié un TIMEOUT.

VI. Gestion des sémaphores.

VI.1 Les sémaphores objets du noyau.

Contrairement aux boîtes à lettres, les sémaphores sont des objets gérés par le noyau au sein duquel elles sont identifiées par un entier compris entre 0 et MAX_SEMAPHORE-1. MAX_SEMAPHORE est une constante du noyau définissant le nombre de sémaphores qu'il doit réserver lors de son initialisation (par MYOS_INIT.)

A chaque sémaphore sont associés une valeur et un statut. La valeur représente le nombre de lectures autorisées. Le statut renseigne si elle est libre ou utilisée.

VI.2 Les opérations.

VI.2.A Création d'une sémaphore: SC-SCREATE.

Avant d'utiliser une sémaphore elle doit être créée par SC_SCREATE. Ce service cherche une sémaphore de libre, la réserve et l'initialise avec la valeur fournie comme paramètre. L'identifiant de la sémaphore est ensuite transmis au programme appelant. Un message d'erreur est retourné si toutes les sémaphores sont réservées.

Pour une recherche rapide d'une sémaphore de libre, les sémaphores sont des éléments d'une liste doublement chaînée (les éléments de la liste sont LISTE_SEMAPHORE [x]). Au niveau de l'implémentation un pointeur LISTE_SEMAPHORE_LIBRE indiquera une sémaphore de libre.

Pseudo-code:

INT SC_SCREATE (INT VALEUR_INITIALE, INT _FAR *ERREUR)

```
SI LISTE_SEMAPHORE_LIBRE NOT =NULL
   IDENTIFIANT_SEMAPHORE=LISTE_SEMAPHORE.IDENTIFIANT
   LISTE_SEMAPHORE[IDENTIFIANT_SEMAPHORE].STATUT=UTILISE
   LISTE_SEMAPHORE[IDENTIFIANT_SEMAPHORE].VALEUR=
                                          VALEUR_INITIALE
   LISTE_SEMAPHORE[IDENTIFIANT_SEMAPHORE].NB_TACHE_EN_ATTENTE=0
   METTRE A JOUR LES POINTEURS DE LISTE DES SEMAPHORES
   ERREUR=RET_OK
   RETOURNE IDENTIFIANT_SEMAPHORE
AUSSI NON
   *ERREUR=ER_NOCB
FIN SI
```

VI.2.B Lecture d'une sémaphore: SC-PEND.

Cette fonction correspond à P(s) défini par Dijkstra. L'identifiant d'une sémaphore est transmis comme paramètre à SC_PEND. La valeur de la sémaphore spécifiée est décrémentée de 1 si elle est strictement positive. Une valeur négative ou nulle suspend la tâche et enregistre la requête dans une file d'attente.

Les deux fonctions suivantes ont été créées pour gérer les opérations sur les listes doublement chaînées:

AJOUTER_ATTENTE_SEMAPHORE (INT SEM_ID, INT TACHE_ID, INT *ERREUR)

Elle insère la requête de la tâche TACHE_ID dans la file d'attente de la sémaphore SEM_ID. Le drapeau "en attente d'une sémaphore est levé" dans le TCB de la tâche TACHE_ID. ERREUR contient l'adresse de la variable destinée à contenir le code d'erreur si la sémaphore est effacée.

ENLEVER_ATTENTE_SEMAPHORE (INT TACHE_ID)

Elle retire de la file d'attente des sémaphores la tâche TACHE_ID.

Pseudo-code:

SC_SPEND (INT SEM_ID, LONG TIMEOUT, INT *ERREUR)

```
SI LISTE_SEMAPHORE[SEM_ID].STATUT=LIBRE
 *ERREUR=ER_ID
AUSSI NON
 *ERREUR=RET_OK
 SI LISTE_SEMAPHORE[SEM_ID].VALEUR=0
    AJOUTER_ATTENTE_SEMAPHORE(SEM_ID,TACHE_EN_COURS,ERREUR)
    SI TIMEOUT>0
      AJOUTER_ATTENTE_DELAY (TACHE_EN_COURS,TIMEOUT,
                             TIMEOUT,ERREUR)
    FIN SI
    SUSPEND_TACHE (TACHE_EN_COURS);
 AUSSI NON
    LISTE_SEMAPHORE[SEM_ID].VALEUR=LISTE_SEMAPHORE[SEM_ID]
                                                  .VALEUR-1
 FIN SI
FIN SI
```

VI.2.C Libération d'une sémaphore: SC-SPOST

Après avoir consommé une unité d'une sémaphore, la tâche la libérera grâce à SC_SPOST. Celle-ci réveille la première tâche en attente de cette sémaphore si il en existe au moins une. Dans le cas contraire, la valeur de la sémaphore est incrémentée de 1.

Pseudo-code:

VOID SC_SPOST (INT SEM_ID, INT *ERREUR)

```
SI LISTE_SEMAPHORE[SEM_ID].STATUT=LIBRE
     *ERREUR=ER_ID
AUSSI NON
   *ERREUR=RET_OK
   SI LISTE_SEMAPHORE[SEM_ID].NB_ATTENTE_SEMAPHORE=0
    LISTE_SEMAPHORE[SEM_ID].VALEUR=LISTE_SEMAPHORE[SEM_ID].
                                                    VALEUR+1
   AUSSI NON
    SOIT NUMERO_TACHE la première tâche en attente de SEM_ID
    ENLEVER_ATTENTE_SEMAPHORE (NUMERO_TACHE)
    SI TACHE[NUMERO_TACHE].ATTENTE_DELAY NOT = NULL
     ENLEVER_ATTENTE_DELAY(NUMERO_TACHE)
    FIN SI
    REPRENDRE(NUMERO_TACHE)
   FIN SI
FIN SI
```

VI.2.D Consultation d'une sémaphore: SC-SINQUIRY.

La valeur d'une sémaphore peut être consultée sans être altérée grâce à SC-SINQUIRY.

Pseudo-code:

INT SC-SINQUIRY (INT SEM_ID, INT *ERREUR)

```
SI LISTE_SEMAPHORE[SEM_ID].STATUT=LIBRE
     *ERREUR=ER_ID
AUSSI NON
     *ERREUR=RET_OK
RETOURNE LISTE_SEMAPHORE[SEM_ID].VALEUR
FIN SI
```

VI.2.E Effacement d'une sémaphore.

La destruction d'une sémaphore consiste à la remettre dans l'état libre. Cette opération est très simple si il n'existe pas de tâche en attente de cette sémaphore. Quelle optique adopter dans le cas contraire? Le choix est laissé à l'utilisateur: SC_SDELETE reçoit MODE comme paramètre. Si celui-ci est nul, la sémaphore n'est détruite que si aucune tâche n'est en attente de celle-ci. Spécifier MODE=1 autorise la destruction de la sémaphore. Les tâches qui l'attendent sont réveillées et reçoivent un message d'erreur "sémaphore détruite."

Pseudo-code:

VOID SC_SDELETE (INT SEM_ID, INT *ERREUR)

```
SI LISTE_SEMAPHORE[SEM_ID].STATUT=LIBRE
 SI (MODE=0 et LISTE_SEMAPHORE[SEM_ID].NB_TACHE_AN_ATTENTE>0)
      *ERREUR=ER_PND
AUSSI NON
      ∀ TACHE_ID en attente de SEM_ID
           RENVOYER CODE D'ERREUR "SEMAPHORE EFFACEE"
           ENLEVER_ATTENTE_SEMAPHORE(TACHE_ID)
           SI (TACHE[TACHE_ID].ATTENTE_DELAY NOT = NULL)
                ENLEVER_ATTENTE_DELAY (TACHE_ID)
           FIN SI
           REPRENDRE (TACHE_ID)
      LISTE_SEMAPHORE[SEM_ID].STATUT=LIBRE
      METTRE A JOUR LA LISTE DES SEMAPHORE
      *ERREUR=RET_OK
FIN SI
```

VI.3 Affectation du noyau minimal.

VI.3.A La structure de données

La structure DESCRIPTION_SEMAPHORE définit les sémaphores:

```
STRUCT DESCRIPTION_SEMAPHORE
 {
      INT IDENTIFIANT
      INT STATUT
      INT VALEUR
      INT NB_TACHE_EN_ATTENTE
      STRUCT DESCRIPTION_SEMAPHORE *PROCHAINE_SEMAPHORE_LIBRE
      STRUCT DESCRIPTION_ATTENTE_SEMAPHORE *ATTENTE_SEMAPHORE
 }

STRUCT DESCRIPTION_SEMAPHORE LISTE_SEMAPHORE[MAX_SEMAPHORE]
```

VI.3.B SC-SCREATE

Lors de la création d'une tâche, le pointeur ATTENTE_SEMAPHORE devra être initialisé:

```
TACHE[TACHE_LIBRE].ATTENTE_SEMAPHORE=NULL
```

VI.3.C SC-TDELETE

Si la tâche victime est en attente d'une sémaphore, elle devra être ôtée de la file d'attente:

```
SI TACHE[TACHE_ID].ATTENTE_SEMAPHORE NOT = NULL
    ENLEVER_ATTENTE_SEMAPHORE(TACHE_ID)
FIN SI
```

VI.3.D MYOS-INIT

A chaque élément de LISTE_SEMAPHORE devra être associé un identifiant:

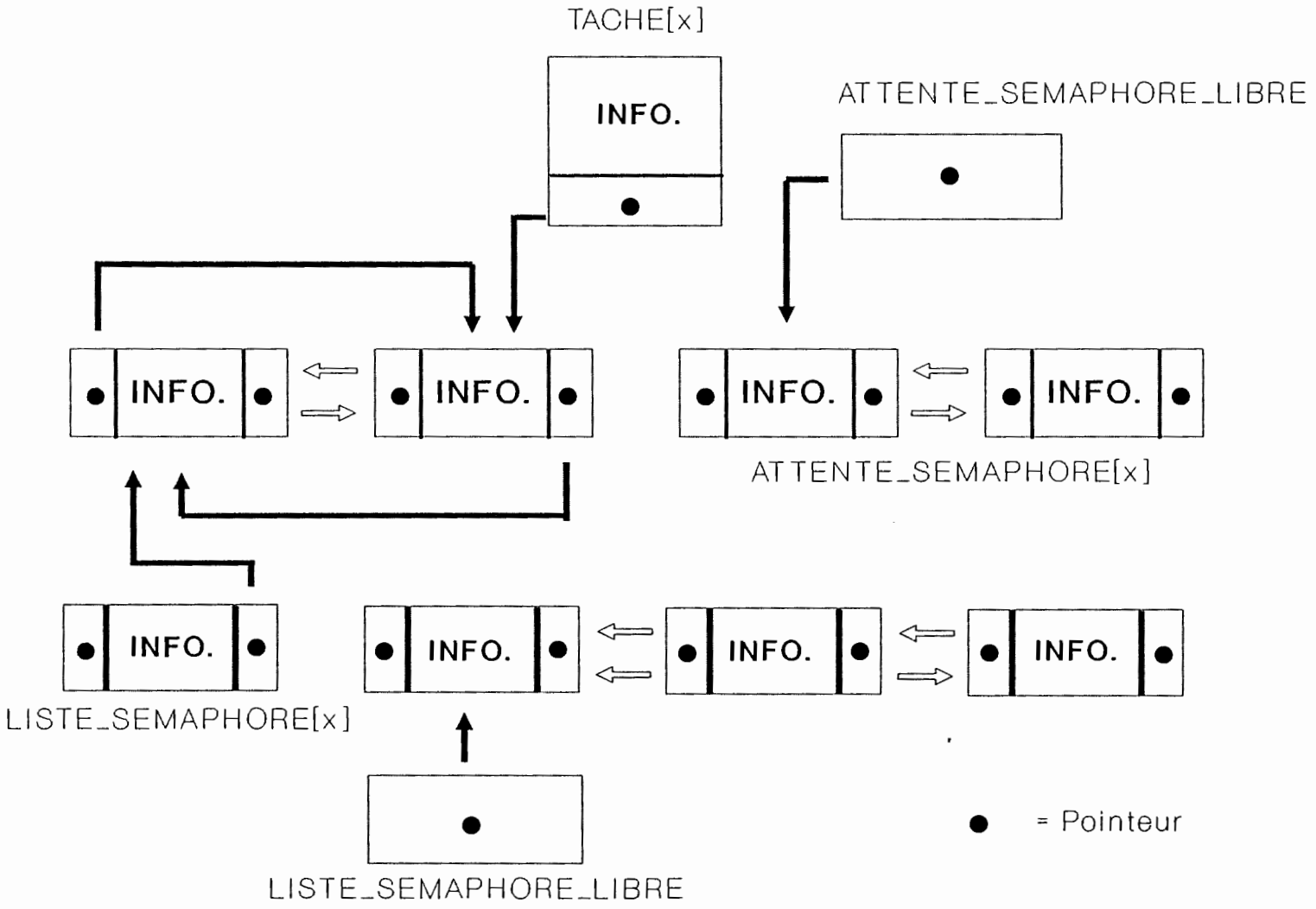
```
POUR 0<= I <= MAX_TACHE-1
    LISTE_SEMAPHORE[I]=I
```

VI.3.E SURVEILLANCE DELAY

Il faudra veiller à enlever de la liste d'attente des sémaphores la tâche dont le timeout vient d'expirer:

```
SI TACHE[NUMERO_TACHE].ATTENTE_SEMAPHORE NOT = NULL
  ENLEVER_ATTENTE_SEMAPHORE (NUMERO_TACHE)
FIN SI
```

Sémaphores et file d'attente.



III ème Partie.

Contenu de la disquette.

Tous les exemples développés dans cette dernière partie sont présents sur la disquette fournie avec ce mémoire. Voici le contenu de cette disquette:

Les applications compilées:[1]

EXEMPLE1.EXE
EXEMPLE2.EXE
EXEMPLE3.EXE
EXEMPLE4.EXE
EXEMPLE5.EXE

EXEMPLE6.EXE
EXEMPLE7.EXE

Ces applications sont directement exécutables.

L'éditeur de texte:

l'éditeur de texte fourni est B.EXE. Il suffit de taper B (éventuellement suivi d'une espace et du nom du fichier à éditer) à partir de MS-DOS pour lancer son exécution. Voici ses 2 commandes essentielles:

<ALT> <X> suivi de <W> pour quitter et sauvé le fichier.
<ALT> <D> supprimer la ligne courante

Pour compiler votre application:

Une commande a été prévue à cet effet: COMPILE. Pour compiler votre application, tapez COMPILE <nom_du_fichier avec l'extension .C>. Votre application est directement disponible (nom_du_fichier .EXE) si votre code source ne comporte pas d'erreurs.

---

[1]Les 5 premiers exemples sont développés dans cette partie.

Exemple 1.

Je voudrais d'abord développer un exemple permettant de visualiser la commutation de tâches. Deux tâches seront créées (TASK_1 et TASK_2): l'une affichera sans cesse à l'écran le chiffre 1, l'autre le chiffre 2. Pour permettre la terminaison du programme, une troisième tâche (TASK_3) sera nécessaire: elle attendra la frappe d'une touche du clavier pour terminer l'application.

```
#include <os.h>              /* mon noyau multitache */
#include <rtl.h>             /* les librairies du noyau */

int erreur;      /* Pour code d'erreur retourne par les fonctions
                    systeme */

/* TASK_1 affiche sans cesse le chiffre 1 à l'écran */

void TASK_1 (void)
{
while (1) myos_putchar ('1');
}

/* TASK_2 affiche sans cesse le chiffre 2 à l'écran */
void TASK_2 (void)
{
while (1) myos_putchar ('2');
}

/*TASK_3 attend la frappe d'une touche puis arrête le programme*/

void TASK_3 (void)
{
char c;           /* pour recevoir le caractere tapé */
c=sc_getc(&erreur);
/* La tache est suspendue jusqu'à la frappe d'une touche */
myos_stop();    /* Arret du scheduler, retour après MYOS_GO() */
}
main()
{
myos_init();
sc_tcreate(TASK_1,1,1,&erreur);  /* Crée la tache 1, prio=1 */
sc_tcreate(TASK_2,2,1,&erreur);  /* Crée la tache 2, prio=1 */
sc_tcreate(TASK_3,3,1,&erreur);  /* Crée la tache 3, prio=1 */
myos_go();                       /* Lance le scheduler */
}
```

Exemple 2.

Je vais à présent illustrer par l'exemple 2 la priorité d'une tâche. Je rappelle que sous MYOS (actuellement) la priorité d'une tâche représente le nombre de tranches de temps qu'elle recevra à chaque fois qu'elle sera mise en exécution par le scheduler.

Les deux tâches de l'exemple 1 sont conservées (TASK_1 et TASK_2.) La troisième tâche (TASK_3) est toujours en attente d'un caractère mais cette fois ci elle filtre le caractère tapé: la frappe du chiffre '2' incrémente la priorité de la tâche 1 tandis que la frappe du chiffre '1' suspend la tache 1 si elle n'était pas suspendue ou la remet dans l'état prêt dans le cas contraire. La frappe de 'Q' arrête le programme.

```
#include <os.h>              /* mon noyau multitache */
#include <rtl.h>             /* les librairies du noyau */

int erreur;      /* Pour code d'erreur retourne par les fonctions
                    systeme */

/* TASK_1 affiche sans cesse le chiffre 1 à l'écran */
void TASK_1 (void)
{
while (1) myos_putchar ('1');
}

/* TASK_2 affiche sans cesse le chiffre 2 à l'écran */
void TASK_2 (void)
{
while (1) myos_putchar ('2');
}

/* TASK_3 attend la frappe d'une touche et l'interprete */
void TASK_3 (void)
{
char c;             /* pour recevoir le caractere tape */
int statut[3];      /* pour recevoir la réponse de SC_TINQUIRY */
while (1)
     {
        c=sc_getc(&erreur);
        /*La tache est suspendue jusqu'a la frappe d'une touche*/
        sc_tinquiry(statut,1,&erreur);
        /* Connaitre l'etat de la tache 1.
              STATUT[0]=1  (TASK_ID)
              STATUT[1]=la priorite de la tache 1
              STATUT[2]=raison de suspension de la tache 1 */
        if (c=='1')
          {
            if (statut[2]!=READY) sc_tresume(1,&erreur);
```

```
            else sc_tsuspend(1,&erreur);
            /* Si la tache etait suspendue alors la reprendre
               aussi non la suspendre */
          };
     if (c=='2')
          {
            sc_tpriority(1,statut[1]+1,&erreur);
            /* incremente la priorite de la tache 1 */
          };
      if (c=='Q' || c=='q') myos_stop(); /* la frappe de Q arrete
le programme */
     }
}
main()
{
myos_init();       /* Initialise les variables du noyau, cree les
TCB ... */
sc_tcreate(TASK_1,1,1,&erreur);  /* Cree la tache 1, prio=1 */
sc_tcreate(TASK_2,2,1,&erreur);  /* Cree la tache 2, prio=1 */
sc_tcreate(TASK_3,3,1,&erreur);  /* Cree la tache 3, prio=1 */
myos_go();                       /*  Lance le scheduler */
}
```

Exemple 3.

Je vais à présent introduire l'utilisation des sémaphores. Deux tâches (TASK_1 et TASK_2) afficheront sans cesse à l'écran respectivement le message "TACHE 1" et "TACHE 2". Pour que ces deux phrases soient écrites en entier, nous considérons l'écran comme une ressource dont l'accès est limité à une tâche. Cette protection est mise en pratique par la création d'une sémaphore initialisée à 1.

Pour illustrer le service de destruction de sémaphore, la troisième tâche (TASK_3) qui est toujours en attente d'un caractère interviendra sur la sémaphore de la façon suivante: la frappe de '1' la détruira tandis que la frappe de '2' en créera une. La frappe de 'Q' met fin au programme.

```
#include <os.h>              /* mon noyau multitache */
#include <rtl.h>             /* les librairies du noyau */

int erreur;/* Pour code d'erreur retourne par les fonctions
              systeme */
int num_sem;      /* l'identifiant d'une semaphore    */
void TASK_1 (void)
{
while (1)
      {
       sc_spend(num_sem,0,&erreur); /*attendre la sémaphore */
       myos_putstr ("TACHE 1");
       sc_spost(num_sem,&erreur);   /* libérer la sémaphore */
      }
}

void TASK_2 (void)
{
while(1)
   {
     sc_spend(num_sem,0,&erreur); /* attendre la sémaphore */
     myos_putstr ("TACHE 2 ");
     sc_spost(num_sem,&erreur);    /* libérer la sémaphore */
   }
}


void TASK_3 (void)
{
char c;           /* pour recevoir le caractere tape */
int statut[3];
while (1)
     {
```

```
      c=sc_getc(&erreur);
                  /* La tache est suspendue jusqu'a la frappe
                    d'une touche */
      if (c=='1')
           {
            sc_sdelete (num_sem,1,&erreur);
            /*Efface la semaphore.Admirez le désordre à l'écran.*/
           };
      if (c=='2')
           {sc_lock(); /*Section critique */
            num_sem=sc_screate (1,&erreur); /* Recree la semaphore
*/
            sc_tdelete(1,&erreur);
            sc_tdelete(2,&erreur);
            sc_tcreate(TASK_1,1,1,&erreur);
            sc_tcreate(TASK_2,2,1,&erreur);
            sc_unlock(); /* Fin de section critique */
           };
         if (c=='Q' || c=='q') myos_stop(); /* la frappe de Q
                                               arrête le prg */
      }
}
main()
{
myos_init();
num_sem=sc_screate(1,&erreur); /* Creation d'une semaphore
                                  initialisee a 1 */
sc_tcreate(TASK_1,1,1,&erreur);  /* Cree la tache 1, prio=1 */
sc_tcreate(TASK_2,2,1,&erreur);  /* Cree la tache 2, prio=1 */
sc_tcreate(TASK_3,3,1,&erreur);  /* Cree la tache 3, prio=1 */
myos_go();                       /*  Lance le scheduler */
        }
```

Exemple 4.

La communication entre tâche est illustrée dans ce quatrième exemple. Les deux tâches d'affichage continuel sont conservées. La troisième tâche (TASK_3) est suspendue jusqu'à la frappe d'un caractère. Si la touche enfoncée est 'Q' alors TASK_3 met fin à l'application. La frappe d'une autre touche crée une quatrième tâche (TASK_4) dont la seule fonction est de déposer un message dans la boîte à lettres. TASK_5 attend un message dans la boîte à lettres. Elle confirmera la réception d'un message en affichant "MESSAGE RECU". La justification de l'existence de TASK_6 est de simuler du processing pour ralentir le système afin que l'utilisateur puisse observer l'exécution (trop rapide sur un 80386).

```
#include <os.h>            /* mon noyau multitache */
#include <rtl.h>           /* les librairies du noyau */

void TASK_4 (void); /* prototype fonction */

int erreur;      /* Pour code d'erreur retourne par les fonctions
                    systeme */
int num_sem;      /* l'identifiant d'une semaphore    */
char boite_a_lettre;
void TASK_1 (void)
{
while (1)
      {
        sc_spend(num_sem,0,&erreur);
        myos_putstr ("TACHE 1 ");
        sc_spost(num_sem,&erreur);
      }
}
void TASK_2 (void)
{
while(1)
   {
     sc_spend(num_sem,0,&erreur);
     myos_putstr ("TACHE 2 ");
     sc_spost(num_sem,&erreur);
   }
}

void TASK_3 (void)
{
char c;            /* pour recevoir le caractere tape */
int statut[3];
while (1)
     {
        c=sc_getc(&erreur);
```

```
                /* La tache est suspendue jusqu'a la frappe
                   d'une touche */
        if (c=='Q' || c=='q') myos_stop(); /*la frappe de Q
                                            arrête le programme */
        sc_tcreate(TASK_4,4,1,&erreur);
      }
}
void TASK_4 (void)
{
 sc_post(&boite_a_lettre,'a',&erreur);
}
void TASK_5 (void)
{
 char message_recu;
while(1)
 {
     sc_pend(&boite_a_lettre,0,&erreur);
     sc_spend(num_sem,0,&erreur);
     myos_putstr("Message reçu ");
     sc_spost (num_sem,&erreur);
 }
}
void TASK_6 (void)
{
debut: ;
/* la meilleure simulation de processing */
goto debut;
}
main()
{
myos_init();        /* Initialise les variables du noyau, cree
                       les TCB ... */
num_sem=sc_screate(1,&erreur); /* Creation d'une semaphore
                                  initialisee a 1 */
sc_tcreate(TASK_1,1,1,&erreur); /* Cree la tache 1, prio=1 */
sc_tcreate(TASK_2,2,1,&erreur); /* Cree la tache 2, prio=1 */
sc_tcreate(TASK_3,3,1,&erreur); /* Cree la tache 3, prio=1 */
sc_tcreate(TASK_5,5,1,&erreur); /* Cree la tache 5, prio=1 */
sc_tcreate(TASK_6,6,4,&erreur); /* Cree la tache 6, prio=4
pour simuler du processing et donc ralentir le autres tâches */
myos_go();                      /*  Lance le scheduler */
        }
```

Exemple 5: le problème des philosophes.

Une table de banquet présente un unique plat de riz autour duquel sont disposées quatre baguettes. Quatre philosophes sont assis à l'entour de cette table. Chacun occupe son temps de façon répétitive à attendre, manger, attendre puis réfléchir. Afin de pouvoir manger, chacun doit d'abord acquérir les deux baguettes qui lui sont adjacentes.

Le fichier suivant contient la déclaration des types et des variables utilisés dans l'application:

Philotyp.h (définitions des types):

```
int erreur;
enum ETAT
    {
       WAITING,
       EATING,
       THINKING,
    };
enum main
    {
           possede_fourchette,
           libre
    };
struct PHILO
    {
         int nom;
         enum main main_gauche;
         enum main main_droite;
         enum ETAT etat;
     };

struct PHILO philosophe[4];
int semaphore_fourchette[4];
char mb_philosophe[4];
```

Pour changer d'état, chaque philosophe attend un stimulus. Celui-ci correspond à la frappe d'une touche: '1' stimule le philosophe 1, '2' stimule le philosophe 2, '3' stimule le philosophe 3 et '4' stimule le philosophe 4.

Pour signaler la présence d'un stimulus, un message sera envoyé au philosophe concerné.

Philosoph.c (le programme) :

```
#include <os.h>
#include <rtl.h>
#include <philotyp.h>
#include <affiche.h> /* mise à jour de l'écran */
#include <graph.h>

void PHILOSOPHE ( void );
void tache_philosophe ( void );
void affiche_philosophe (int numero_philosophe,struct PHILO
*ptr_philo);
void enleve_fourchette (int numero_fourchette);
void pose_fourchette (int numero_fourchette);

main()
{int compteur;
myos_init();
_clearscreen (_GCLEARSCREEN);
sc_tcreate (PHILOSOPHE,1,1,&erreur);
for (compteur=0;compteur<=3;compteur++)
     mb_philosophe[compteur]='\0';/*vider les boites à lettres */
for (compteur=0;compteur<=3;compteur++)
   {
     semaphore_fourchette[compteur]=sc_screate(1,&erreur);
     /* chaque fourchette est une sémaphore */
     pose_fourchette (compteur);
   }
for (compteur=2;compteur<=5;compteur++)
          sc_tcreate(tache_philosophe,compteur,1,&erreur);
          /* Une tâche par philosophe */
myos_go();
};

void  PHILOSOPHE ( void )
{
char c;
while (1)
      {
     c=sc_getc(&erreur);
     if (c=='1') sc_post (&mb_philosophe[0],'1',&erreur);
     if (c=='2') sc_post (&mb_philosophe[1],'1',&erreur);
     if (c=='3') sc_post (&mb_philosophe[2],'1',&erreur);
     if (c=='4') sc_post (&mb_philosophe[3],'1',&erreur);
     if (c=='Q') myos_stop();
      }
}
void tache_philosophe ( void )
{
struct PHILO philosophe;
int fourchette_gauche,fourchette_droite;
int info_tache[3];
```

```
sc_tinquiry(info_tache,0,&erreur);
fourchette_gauche=info_tache[0]-2;
fourchette_droite=fourchette_gauche+1;
if (fourchette_droite==4) fourchette_droite=0;
philosophe.nom=fourchette_gauche;
philosophe.etat=WAITING;
philosophe.main_gauche=libre;
philosophe.main_droite=libre;
affiche_philosophe(philosophe.nom,&philosophe);
while(1)
     {
       sc_pend (&mb_philosophe[philosophe.nom],0,&erreur);
     sc_spend(semaphore_fourchette[fourchette_gauche],0,&erreur);
       philosophe.main_gauche=possede_fourchette;
       affiche_philosophe(philosophe.nom,&philosophe);
       enleve_fourchette (fourchette_gauche);
       sc_pend (&mb_philosophe[philosophe.nom],0,&erreur);

sc_spend(semaphore_fourchette[fourchette_droite],0,&erreur);
       philosophe.main_droite=possede_fourchette;
       enleve_fourchette (fourchette_droite);
       philosophe.etat=EATING;
       affiche_philosophe(philosophe.nom,&philosophe);

       sc_pend (&mb_philosophe[philosophe.nom],0,&erreur);
       pose_fourchette(fourchette_gauche);
       sc_spost(semaphore_fourchette[fourchette_gauche],&erreur);
       philosophe.etat=WAITING;
       philosophe.main_gauche=libre;
       affiche_philosophe (philosophe.nom,&philosophe);

       sc_pend (&mb_philosophe[philosophe.nom],0,&erreur);
       pose_fourchette(fourchette_droite);
       sc_spost(semaphore_fourchette[fourchette_droite],&erreur);
       philosophe.etat=THINKING;
       philosophe.main_droite=libre;
       affiche_philosophe (philosophe.nom,&philosophe);

     }
}
```

Conclusion.

Nous voici arrivés au terme de ce mémoire et nous avons rempli notre cahier des charges du départ: après en avoir défini les concepts, nous avons réalisé un noyau multitâche monoprocesseur. Le lecteur aura ainsi abordé la concurrence sous deux aspects: de la **réalisation** d'un noyau à **l'écriture** de programmes concurrents.

Personnellement, la conception de MYOS m'a appris énormément de choses. En dehors de la réalisation des primitives du noyau, c'est la conception de programmes concurrents qui m'a sensibilisé: à plusieurs reprises j'ai mis en doute le bon fonctionnement de MYOS suite aux résultats erronés de mon application. Après analyse, c'est la **conception** du programme concurrent qui s'est avérée en être la cause: absence du signalement de la fin d'une section critique, apparition de deadlock ...

Concernant MYOS, il sera prochainement optimisé au niveau du temps de latence de certaines primitives du noyau. De nouveaux services lui seront aussi adjoints : ceux pour la gestion des événements et de la communication

Deux continuations peuvent être proposées à cette étude. D'une part, maintenant que nous possédons des systèmes multitâche comme support à nos programmes concurrents, ceux-ci doivent être conçus autrement. A cet égard, il serait intéressant de réfléchir sur leurs méthodes de **conception** (je pense en autre au langage UNITY avec toute la méthode qu'il propose). D'autre part, la généralisation à plusieurs processeurs pourrait être la base de la conception d'un système multitâche **multiprocesseur**.

Annexes.

Table des matières.

Introduction.

Le lecteur trouvera dans ces annexes la description de toutes les fonctions de MYOS. A cet effet le canevas suivant sera adopté:

- **précondition(s)** : ce qui doit être vrai avant l'appel de la fonction pour que celle-ci s'exécute sans code d'erreur en retour

- **spécification**: ce que fait la fonction

- **codes d'erreur**: la liste des codes d'erreur que peut retourner la fonction.

- **exemple d'appel**: une illustration de l'appel du service en langage C.

Les services de RTL (Real Time Library) seront développés de la même façon. L'existence de cette librairie se justifie par le fait que les fonctions des librairies fournies avec les compilateurs C ne peuvent être utilisées avec MYOS puisqu'elles ne sont pas conçues pour être exécutées de façon concurrente.

Enfin, à titre d'illustration, le code source de quelques services du noyau est fourni dans la troisième partie.

I. Les services de MYOS.

Pour chaque fonction de MYOS, ce chapitre décrit leur(s) précondition(s), leur spécification, les codes d'erreur retournés et un exemple en langage C. Il reprend aussi la liste de tous les codes d'erreur et de tous les statuts de tâche. Le lecteur pourra au terme de ce chapitre écrire des applications multitâche avec le noyau fourni[1] avec ce mémoire.

ATTENTION: pour plus de clarté, le nom de chaque fonction est écrit en majuscule mais lors de l'écriture d'une application, le nom de chaque service doit être en **minuscule** !!!

[1] Sa version compilée uniquement.

La gestion des tâches

void **SC_TCREATE** (void ***ADRESSE_DEPART** ,int **ID**, int **PRIO** ,int ***ERREUR**)

Préconditions:

0< ID <= MAX_TACHE et identifie une tâche NON-EXISTANTE.
0< PRIO <= 255
ADRESSE_DEPART est l'adresse de la première instruction de la tâche à lancer. Il s'agit du nom de la fonction C à exécuter.

Spécification:

Crée le TCB (Task Control Block) pour la tâche identifiée par ID. PRIO détermine le nombre de tranches de temps CPU allouées à la tâche lorsqu'elle a le contrôle. La tâche crée est dans l'état prêt après sa création.

Codes d'erreur:

RET_OK: pas d'erreur, la tâche est créée.
ER_TID: mauvaise valeur de ID
ER_TCB: plus de TCB de libre.

Exemple d'appel:

```
int erreur;
void PROCESS_1 ( VOID )
        {
                myos_putstr('Hello!');
        };
 main ()
        {
                SC_TCREATE (PROCESS_1,2,1,&ERREUR)
        }
```

void **SC_TSUSPEND** (int **ID**, int ***ERREUR**);

Précondition:

0 <=ID <= MAX_TACHE et ID identifie une tâche existante.

Spécification:

Cette fonction suspend la tâche identifiée par ID. Si ID est nul, alors la tâche se suspend elle-même.

Codes d'erreur:

RET_OK: pas d'erreur, la tâche ID est suspendue.
ER_TID : mauvaise valeur de ID.

Exemple d'appel:

```
SC_TSUSPEND (1,&erreur) /* suspend le tache 1   */
SC_TSUSPEND (0,&erreur) /* se suspend elle-même */
```

void **SC_TRESUME** (int **ID**, int **ERREUR**);

Précondition:

0 < ID <= MAX_TACHE et identifie une tâche existante.

Spécification:

Annule l'effet d'un SC_TSUSPEND sur la tâche ID. Si il n'existe pas d'autres causes de suspension que le SC_TSUSPEND, la tâche ID est remise dans l'état prêt.

Codes d'erreur:

RET_OK: pas d'erreur, la tâche ID est remise dans l'état prêt si il n'existe pas d'autres causes de suspension que SC_TSUSPEND.
ER_TID : tâche inexistante.

Exemple d'appel:

```
SC_TRESUME (1,&erreur)   /* Reprend la tâche 1 */
```

void **SC_TINQUIRY** (int ***INFO**, int **ID**, int ***ERREUR**);

Précondition:

0<= ID <= MAX_TACHE et identifie une tache existante. INFO est un tableau de 3 entiers.

Spécification:

Renvoye le statut d'une tâche. Si ID est nul, elle renvoye le statut de la tâche en exécution.

INFO contient:

INFO[0]: ID
INFO[1]: priorité
INFO[2]: statut

Codes d'erreur:

RET_OK: pas d'erreur; INFO contient les informations concernant la tâche ID.
ER_TID : tache inexistante.

Exemple d'appel:

```
int INFO[3];
SC_TINQUIRY (INFO, 2, &erreur);
myos_printf("Priorite de la tache 2 %d",info[1]);
```

void **SC_TPRIORITY** (int **ID** , int **PRIO**, int ***ERREUR**);

Préconditions:

0<= ID <= MAX_TACHE et identifie une tâche existante.
0<= PRIO <= 255

Spécification:

Change la priorité d'une tâche existante. Si ID est nul alors la tâche modifie sa propre priorité.

Codes d'erreur:

RET_OK: pas d'erreur, la priorité de la tâche ID est maintenant PRIO.
RET_TID:la tâche est inexistante.

Exemple d'appel:

```
int erreur;
SC_TPRIORITY(0,2,&erreur); /* Se donne une priorité 2 */
```

void **SC_TDELETE** (int **ID**,int ***ERREUR**);

Précondition:

0<= ID <=MAX_TACHE et identifie une tâche existante

Spécification:

Tue la tâche identifiée par ID. Si ID est nul, la tâche se suicide.

Codes d'erreur:

RET_OK: pas d'erreur, la tâche ID est tuée.
ER_TID: tâche inexistante.

Exemple d'appel:

```
SC_TDELETE(4,&erreur); /* Tue le tâche 4 */
SC_TDELETE(0,&erreur); /* Se suicide      */
```

Fonctions de blocage et de déblocage de l'ordonnancement

void **SC_LOCK** (void)

Précondition:

Aucune

Spécification:

Bloque le scheduler. La tâche qui a la contrôle le garde jusqu'au prochain SC_UNLOCK. Elle ne sera jamais préemptée.

Codes d'erreur:

Aucun

Exemple d'appel:

```
SC_LOCK(); /* Bloque le scheduler */
```

void **SC_UNLOCK** (void);

Précondition:

Aucune

Spécification:

Débloque le scheduler. La tâche qui a la contrôle peut à nouveau être préemptée.

ATTENTION: le kernel tient à jour un compteur d'imbrication de SC_LOCK. Ce compteur est incrémenté à chaque appel de SC_LOCK et décrémenté à chaque appel de SC_UNLOCK. Le scheduler est débloqué lorsque ce compteur est nul.

Codes d'erreur:

Aucun

Exemple d'appel:

```
SC_LOCK();
      {
            .
            .
            /* Section critique */
            .
       };
SC_UNLOCK();
```

Fonctions concernant les MAILBOXES

void **SC_POST** (char ***MAILBOX**, char **MESSAGE**, int ***ERREUR**);

Préconditions:

MAILBOX pointe vers un caractère nul.
0< MESSAGE <=255.

Spécification:

Poste MESSAGE dans la boîte à lettres d'adresse MAILBOX. Si des tâches sont en attente d'un message en provenance de la boîte à lettres d'adresse MAILBOX, la première d'entre elles reçoit directement le message et est remise dans l'état prêt si il n'existe plus de conditions de blocage.

Codes d'erreur:

RET_OK: pas d'erreur, le message est posté.
ER_MIU: la boîte à lettres est pleine.
ER_ZMW: message nul

Exemple d'appel:

```
char BOITE_A_LETTRE_1;
int erreur;
SC_POST (&BOITE_A_LETTRE_1, 'A', &erreur)
/* poste le message 'A' dans la boîte d'adresse BOITE_A_LETTRE _1*/
```

char **SC_PEND** (char ***MAILBOX**, long **TIMEOUT**, int ***ERREUR**);

Précondition:

Aucune

Spécification:

Si TIMEOUT=0

La tâche est suspendue jusqu'à ce qu'un message soit posté dans la boîte à lettres d'adresse MAILBOX. SC_PEND renvoye le message reçu.

Si TIMEOUT>0

La tâche est suspendue au maximum pendant le nombre de TICKS défini par TIMEOUT. Si un message est reçu avant cette échéance, il sera retourné par la fonction. Si le délai est dépassé, une erreur de timeout est renvoyé.

Codes d'erreur:

RET_OK: pas d'erreur, message bien reçu.
Si TIMEOUT>0
ER_TMO: pas de message reçu dans le délai prévu.

Exemple d'appel:

```
char BOITE_A_LETTRE_1,REPONSE;
int erreur;
reponse=SC_PEND (&BOITE_A_LETTRE_1,0,&erreur);
/* Attend un message dans BOITE_A_LETTRE_1 */
reponse=SC_PEND (&BOITE_A_LETTRE_2,20,&erreur);
/* Attend un message dans BOITE_A_LETTRE_1 pendant 20 TICKS */
```

Fonction concernant le CLAVIER

char **SC_GETC** (int ***ERREUR**);

Précondition:

Aucune

Spécification:

La tâche est suspendue jusqu'à l'arrivée d'un caractère du clavier. La fonction SC_GETC renvoye le caractère tapé.

Codes d'erreur:

RET_OK: pas d'erreur, le caractère tapé est retourné
ER_CIU: une autre tâche attend déjà un caractère du clavier.

Exemple d'appel:

```
char a;
int erreur;
a=SC_GETC(&erreur);  /* Attend un caractère */
```

Fonctions concernant les SEMAPHORES

int **SC_SCREATE** (int **VALEUR_INITIALE**, int ***ERREUR**);

Précondition:

Le système possède encore de la place mémoire pour créer une sémaphore créée

Spécification:

Crée une sémaphore avec une valeur initiale. La fonction retourne l'identifiant de la sémaphore.

Codes d'erreur:

RET_OK: pas d'erreur, sémaphore créée et initialisée avec VALEUR_INITIALE.
ER_NOCB: plus de place pour une sémaphore.

Exemple d'appel:

```
int numero_sem;
numero_sem=SC_SCREATE(2,&erreur); /* Crée une sémaphore de
                                     valeur initiale 2.
                                     numero_sem contiendra l'id
                                     de la sémaphore créée */
```

void **SC_SPEND** (int **ID**, long **TIMEOUT**, int ***ERREUR**);

Préconditions:

0 <= ID <=MAX_SEMAPHORE-1et identifie une sémaphore existante.
0 <= TIMEOUT <=65535

Spécification:

Si la sémaphore identifiée par ID est nulle, la tâche est suspendue et mise en attente de cette sémaphore (FIFO) ou jusqu'à l'expiration du délai (TIMEOUT>0)

Si la sémaphore identifiée par ID n'est pas nulle, la sémaphore est décrémentée.

Codes d'erreur:

RET_OK: pas d'erreur, la valeur de la sémaphore est décrémentée.
ER_TMO: délai expiré, la sémaphore n'est pas décrémentée.
ER_ID : sémaphore inexistante.
ER_DEL: sémaphore effacée.

Exemple d'appel:

```
SC_SPEND (numero_sem,0,&erreur) /*Attend la libération de la
                                  sémaphore d'id numero_sem */
```

void **SC_SDELETE** (int **ID**, int **MODE**, int ***ERREUR**);

Préconditions:

0<= ID <= MAX_SEMAPHORE-1 et identifie une sémaphore existante.
MODE = 0 ou MODE = 1

Spécification:

Si MODE=0

la sémaphore est effacée si aucune tâche n'est en attente de cette sémaphore.

Si MODE=1

la sémaphore est effacée. Toutes les tâches en attente de cette sémaphore sont remises dans l'état prêt et un code d'erreur "sémaphore effacée" est renvoyé aux tâches en attente de cette sémaphore.

Codes d'erreur:

RET_OK: pas d'erreur, sémaphore effacée.
ER_ID : sémaphore inexistante

Si MODE=0
ER_PND: des tâches sont en attente de cette sémaphore.

Exemple d'appel:

```
SC_SDELETE (numero_sem, 0, &erreur); /* Efface la sémaphore d'id
                                        numero_sem. */
```

int **SC_SINQUIRY** (int **ID**, int ***ERREUR**);

Précondition:

0<= ID <= MAX_SEMAPHORE-1 et identifie une sémaphore existante.

Spécification:

Retourne le valeur de la sémaphore identifiée par ID.

Codes d'erreur:

RET_OK: pas d'erreur.
ER_ID : sémaphore inexistante.

Exemple d'appel:

```
int compteur;
compteur=SC_TINQUIRY(numero_sem,&erreur)
/*Retourne dans compteur la valeur de la sémaphore  numero_sem */
```

void **SC_SPOST** (int **ID**, int ***ERREUR**);

Précondition:

0<= ID <=MAX_SEMAPHORE-1 et identifie une sémaphore existante.

Spécification:

Incrémente la sémaphore identifiée par ID. Si des tâches sont en attente de cette sémaphore, la première est remise dans l'état prêt.

Codes d'erreur:

RET_OK: pas d'erreur, sémaphore incrémentée.
ER_ID : sémaphore inexistante.
ER_OVF: overflow, valeur de sémaphore >65535

Exemple d'appel:

```
SC_SPOST (numero_sem, &erreur) /*Incrémente la sémaphore
                                 numéro_sem et remet à  l'état prêt
                                 la première tâche en attente de cette
                                 sémaphore (si c'est la  seule cause
                                 de suspension)*/
```

Fonction de gestion du temps

void **SC_DELAY** (long **TIMEOUT**);

Précondition:

Aucune.

Spécification:

La tâche est suspendue pendant un nombre de ticks déterminé par TIMEOUT.

Codes d'erreur:

Aucun.

Exemple d'appel:

```
SC_DELAY ( 100 ) /* S'endort pendant 100 TICKS */
```

Fonctions propres à MYOS.

void **MYOS_INIT** (void)

Précondition:

Aucune.

Spécification:

Initialise toutes les variables du kernel (TCB,files d'attente,stack ...)

Codes d'erreur:

Aucun.

Exemple d'appel:

```
MYOS_INIT();
```

void **MYOS_GO** (void)

Précondition:

Les variables du kernel ont été initialisées par l'appel de MYOS_INIT();

Spécification:

Lance le scheduler si il existe au moins une tâche de prête.

Codes d'erreur:

Aucun.

Exemple d'appel:

MYOS_GO()

void MYOS_STOP (void)

Précondition:

Le scheduler a été lancé par MYOS_GO().

Spécification:

Arrête le scheduler. Le contrôle est rendu au programme ayant lancé le scheduler via MYOS_GO().

Codes d'erreur:

Aucun.

Exemple d'appel:

MYOS_STOP()

Liste des codes d'erreur.

Codes:	Signification:
RET_OK	Retour avec succès
ER_TID	Erreur d'identification de tâche
ER_TCB	Plus de TCB de libre
ER_MIU	MAILBOX déjà utilisée
ER_ZMW	Message nul
ER_TMO	Timeout
ER_NMP	Pas de message présent
ER_NOCB	Plus de control block de libre
ER_ID	Mauvais identificateur
ER_PND	Tâche en attente d'un sémaphore
ER_DEL	sémaphore effacée
ER_OVF	Débordement
ER_CIU	Canal réservé

Codes de statut d'une tache

Constante	Raison	Appel système
EX_SUS	Suspendue explicitement	SC_TSUPSEND
MB_SUS	En attente de message	SC_PEND
CH_SUS	En attente de caractère	SC_GETC
DL_SUS	En attente de délais	SC_DELAY
SM_SUS	En attente de sémaphore	SC_SPEND

II. La libraire annexe: RTL

Toutes les libraires fournies avec le compilateur C ne sont plus valables en multitâche puisque leurs fonctions sont écrites dans une optique de programmation séquentielle: certaines d'entre elles modifient la valeur de la pile, bloquent les interruptions ou font appel aux fonctions du BIOS (celles-ci bloquent aussi les interruptions et souvent modifient le registre de pile SP par affectation)

Cinq fonctions ont été réécrites afin que l'utilisateur puisse bénéficier de quelques fonctions de sortie à l'écran.

void **MYOS_PUTCHAR** (char **CARACTERE**)

Précondition:

Aucune

Spécification:

Le contenu de la variable CARACTERE est affiché à l'écran.

Codes d'erreur:

Aucun.

Exemple d'appel:

```
char c;
c='A'
myos_putchar (c);
```

void **MYOS_PUTSTR** (char **STRING**[])

Précondition:

String est une chaîne de caractères terminée par le code ASCII nul ("\0".)

Spécification:

STRING est affiché à l'écran à la position courante du curseur

Codes d'erreur:

Aucun

Exemple d'appel:

```
char c[]="SALUT";
myos_putstr ("bonjour"); /* pas besoin de rajouter le code \0, le
                            compilateur s'en charge */
myos_putstr (c);
```

void **MYOS_PRINTF** (char ***FORMAT**, ...)

Précondition:

aucune

Spécification:

Cette fonction est identique au PRINTF qui **NE PEUT ETRE UTILISEE** en multitâche !!!

Codes d'erreur:

Aucun

Exemple d'appel:

```
int nb_clients,tableau[10],compteur;
myos_printf("Nombre de clients: %d ",nb_clients);
for (compteur=0;compteur<=9;compteur++)
      myos_printf ("Tableau[%d]=%d",compteur,tableau[compteur]);
```

void **MYOS_BEEP_ON** (void)

Précondition:

Aucune

Spécification:

Active le générateur de son du PC

Codes d'erreur

Aucun

Exemple d'appel:

```
myos_printf("ERREUR !!!");
myos_beep_on();
```

void **MYOS_BEEP_OFF** (void)

Précondition:

Aucune

Spécification:

Désactive le générateur de son du PC

Codes d'erreur

Aucun

Exemple d'appel:

myos_beep_off();

III. Code source

A titre d'illustration, voici le code source de quelques primitives du noyau:

Le SCHEDULER

```
void _interrupt _far scheduler(void)
{
struct description_tache _far *ptr_tache;
void _far *variable_tampon;
_DISABLE;
ptr_tache=&tache[tache_en_cours];
_asm mov  variable_tampon,bp
tache[tache_en_cours].stack=variable_tampon ;
if ( dans_section_critique==0 && nb_tache_prete>0 &&
temoin_de_presence==0)
     {
          if (ptr_tache->nb_tranche_deja_obtenue>= ptr_tache-
>priorite ||
            reschedule==1 )
           {
             ptr_tache->nb_tranche_deja_obtenue=0;
             if (reschedule==1) ptr_tache-
>statut=statut_rescheduling;
               else  ptr_tache->statut=prete;
                   tache_en_cours=tache_suivante();
           }
        ptr_tache=&tache[tache_en_cours];
        ptr_tache->statut=execution;
        ptr_tache->nb_tranche_deja_obtenue++;
         }
else if (temoin_de_presence==1) loupe_interrupt=1;
if (reschedule==0)
   {
     (*oldtimer)();
     surveillance_delay();
   }
  else reschedule=0;
variable_tampon=tache[tache_en_cours].stack;
_asm mov bp,  variable_tampon ;
_ENABLE ;
};
```

SUSPEND_TACHE

```
void _far suspend_tache (int id)
{
 _DISABLE
     if (tache[id].statut==prete || tache[id].statut==execution)
          nb_tache_prete--;
     tache[id].statut=suspendue;
     if (id==tache_en_cours)
     {
       if (nb_tache_prete==0)
          {
          idle_task();
          reschedule=1;
          if (derniere_tache_rendue_prete==tache_en_cours)
                 {
                  statut_rescheduling=prete;/*pour ne pas fausser
                                              timer*/
                 }

          else
               {
                reschedule=1;
                statut_rescheduling=suspendue;
               };
          sc_unlock();
          scheduler();
          }
        else
          {
             reschedule=1;
             statut_rescheduling=suspendue;
             scheduler();
          }
     }
 _ENABLE
}
```

SC_PEND

```
char _far sc_pend (char _far *mailbox, long timeout, int _far
*erreur)
{char resultat;
 struct attente_mb          _far *ptr1_att_mb,
                            _far *ptr2_att_mb,
                            _far *ptr3_att_mb;
 _DISABLE
if (*mailbox!=0)
{
   sc_lock();
   resultat=*mailbox;
   *mailbox=0;
   *erreur=RET_OK;
   sc_unlock();
}
  else
     {
         sc_lock();
tache[tache_en_cours].attente_mailbox=attente_mailbox_libre;
         tache[tache_en_cours].suspension|=MB_SUS;
         attente_mailbox_libre->tache_id=tache_en_cours;
         attente_mailbox_libre->adresse_boite_a_lettre=mailbox;
         attente_mailbox_libre->adresse_reception=&resultat;
      if (premier_attente_mb==NULL)
            {
                  ptr1_att_mb=attente_mailbox_libre;
                  attente_mailbox_libre=attente_mailbox_libre-
>prochaine_attente_mb;
             ptr1_att_mb->prochaine_attente_mb=ptr1_att_mb;
                  ptr1_att_mb-
>precedente_attente_mb=ptr1_att_mb;
                  premier_attente_mb=ptr1_att_mb;
             }
         else
             {
                ptr1_att_mb=attente_mailbox_libre;
                attente_mailbox_libre=attente_mailbox_libre-
>prochaine_attente_mb;
                ptr2_att_mb=premier_attente_mb;
                ptr3_att_mb=premier_attente_mb-
>precedente_attente_mb;

                ptr1_att_mb->prochaine_attente_mb=ptr2_att_mb;
                ptr1_att_mb->precedente_attente_mb=ptr3_att_mb;
                ptr2_att_mb->precedente_attente_mb=ptr1_att_mb;
                ptr3_att_mb->prochaine_attente_mb=ptr1_att_mb;
             }
         _DISABLE;
         sc_unlock();
```

```
          *erreur=RET_OK;
          if (timeout!=0) ajoute_timeout (timeout,erreur);
          suspend_tache(tache_en_cours);
      }
_ENABLE
return resultat;
```

SC_POST

```
void _far sc_post (char _far *mailbox, char message, int _far
*erreur)
{int  compteur, numero_tache, trouve;
 struct attente_mb _far *ptr1_att_mb,_far *ptr2_att_mb;
 _DISABLE;
 sc_lock();
 trouve=0;
 if (message==0)       *erreur=ER_ZMW;
 else
   if (*mailbox!=0) *erreur=ER_MIU;
     else
       {
         *erreur=RET_OK;
           {
              ptr1_att_mb=premier_attente_mb;
              ptr2_att_mb=NULL;
              while(ptr1_att_mb!=ptr2_att_mb && trouve==0)
                {
                   if (ptr1_att_mb-
>adresse_boite_a_lettre==mailbox)
                     {
                       *(ptr1_att_mb->adresse_reception)=message;
                       enlever_attente_mb(ptr1_att_mb->tache_id);
                       if (tache[ptr1_att_mb-
>tache_id].attente_delay!=NULL)
                          enlever_attente_delay(ptr1_att_mb-
>tache_id);
                       reprendre(ptr1_att_mb->tache_id);
                       trouve=1;
                     }
                   else
                       {
                          ptr2_att_mb=premier_attente_mb;
                          ptr1_att_mb=ptr1_att_mb-
>prochaine_attente_mb;
                       }
                }
           }
           if (trouve==0)    *mailbox=message;
       }
 sc_unlock();
 _ENABLE;
}
```

Bibliographie.

M. TISCHER, la bible du PC, Micro application, PARIS, 1989

A. SCHIPER, programmation concurrente, presse polytechniques romandes Lausanne, Lausanne, 1986.

S. KRAKOWIAK, principe des systèmes d'exploitation des ordinateurs, DUNOD, 1986.

D. TSCHIRHART, commande en temps réel, DUNOD, 1990.

Cours d'apprentissage au système temps réel VRTX32, READY SYSTEMS FRANCE (PARIS).

J. RAMAEKERS, syllabus 'systèmes d'exploitation' dans le cadre du cours de 2ème licence, NAMUR, 1990.

JAMES W. COFFRON, Programmation du 8086-8088, SYBEX, 1984